Developing Native Extensions for ADOBE® AIR®

Contents

Chapter 1: Introducing native extensions for Adobe AIR

Native Extensions for Adobe AIR are code libraries that contain native code wrapped with an ActionScript API. You can use native extensions in an AIR application to access platform features not supported by AIR, to benefit from native-code-level performance for critical algorithms, and to reuse existing native code libraries.

About native extensions

What is Adobe AIR?

Adobe® AIR® is a cross-operating system runtime that allows content developers to build rich Internet applications (RIAs). The developers can deploy the RIAs to the desktop, mobile devices, and digital home devices. AIR applications can be built using Adobe® Flex® and Adobe® Flash® (SWF-based) and also with HTML, JavaScript, and Ajax (HTML-based). For more information about the Adobe Flash Platform tools that you can use to build AIR applications, see Adobe Flash Platform tools for AIR development in *Building Adobe AIR Applications*.

What is Adobe ActionScript?

SWF-based AIR applications can use Adobe ActionScript® 3.0. ActionScript 3.0 is an object-oriented language that can add interactivity and data-handling to RIAs. For more information about the language, see *Learning ActionScript 3.0* and *ActionScript 3.0 Developer's Guide*.

ActionScript provides many built-in classes. For example, MovieClip, Array, and NetConnection are built-in ActionScript classes. Additionally, a content developer can create application-specific classes. Sometimes an application-specific class derives from a built-in class.

The runtime executes the code in ActionScript classes. The runtime also executes JavaScript code that is used in HTML-based applications.

What is a native extension?

A native extension is a combination of:

- ActionScript classes.
- Native code. Native code is defined here as code that executes outside the runtime. For example, code that you write in C is native code. On some platforms, Java code is supported in extensions. For the purpose of this documentation, this is also considered “native” code.

Reasons to write a native extension include the following:

- A native code implementation provides access to device-specific features. These device-specific features are not available in the built-in ActionScript classes, and are not possible to implement in application-specific ActionScript classes. The native code implementation can provide such functionality because it has access to device-specific hardware and software.
- A native code implementation can sometimes be faster than an implementation that uses only ActionScript.

- A native code implementation allows you to reuse existing code.

For example, you could create a native extension that allows an application to do the following:

- make a mobile device vibrate.
- interact with device-specific libraries and features.

When you have finished your ActionScript and native implementations, you package your extension. Then, an AIR application developer can use the package to call your extension's ActionScript APIs to execute device-specific functionality. The extension runs in the same process as the AIR application.

Native extensions versus the NativeProcess ActionScript class

ActionScript 3.0 provides a NativeProcess class. This class lets an AIR application execute native processes on the host operating system. This capability is similar to native extensions, which provide access to device-specific features and libraries. When deciding on using the NativeProcess class versus creating a native extension, consider the following:

- Only the `extendedDesktop` AIR profile supports the NativeProcess class. Therefore, for applications with the AIR profiles `mobileDevice` and `extendedMobileDevice`, native extensions are the only choice.
- Native extension developers often provide native implementations for various platforms, but the ActionScript API they provide is typically the same across platforms. When using the NativeProcess class, ActionScript code to start the native process can vary among the different platforms.
- The NativeProcess class starts a separate process, whereas a native extension runs in the same process as the AIR application. Therefore, if you are concerned about code crashing, using the NativeProcess class is safer. However, the separate process means that you possibly have interprocess communication handling to implement.

Native extensions versus ActionScript class libraries (SWC files)

The most important difference between a native extension and a SWC file is that the SWC file contains no native code. Therefore, if you determine that you can accomplish your goal without native code, use a SWC file rather than a native extension.

More Help topics

About SWC files

Supported devices

You can create native extensions for the following devices:

- Android devices, starting with AIR 3 and Android 2.2.
- iOS devices, starting with AIR 3 and iOS 4.0
- iOS Simulator, starting with AIR 3.3
- Blackberry PlayBook, starting with AIR 2.7
- Windows desktop devices that support AIR 3.0
- Mac OS X desktop devices that support AIR 3.0

An extension can target multiple platforms. For more information, see "Targeting multiple platforms" on page 5.

Supported device profiles

The following AIR profiles support native extensions:

- `extendedDesktop`, starting in AIR 3.0
- `mobileDevice`, starting in AIR 3.0

More Help topics

AIR profile support

Native extensions architecture

Architecture overview

AIR allows an extension to do the following:

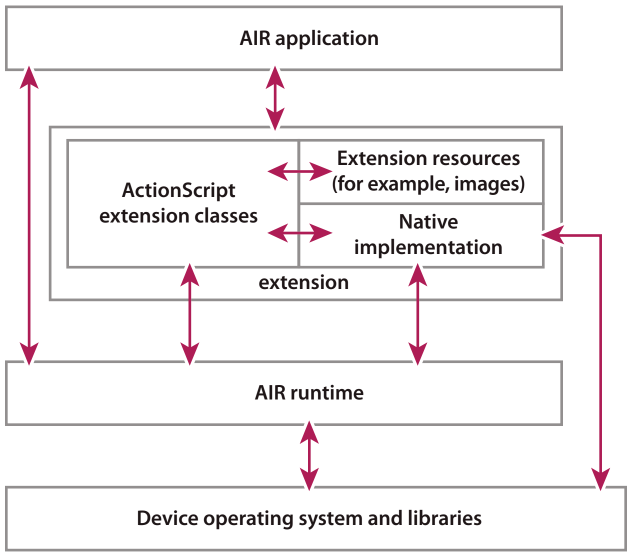
- Call functions implemented in native code from ActionScript.
- Share data between ActionScript and the native code.
- Dispatch events from the native code to ActionScript.

When you create a native extension, you provide the following:

- ActionScript extension classes that you define. These ActionScript classes use the built-in ActionScript APIs that allow access to and data exchange with native code.
- A native code implementation. The native code uses native code APIs that allow access to and data exchange with your ActionScript extension classes.
- Resources, such as images, that the ActionScript extension class or the native code uses.

Your native extension can target multiple platforms. When it does, you can provide a different set of ActionScript extension classes and a different native code implementation for each target platform. For more information, see "Targeting multiple platforms" on page 5.

The following illustration shows the interactions between the native extension, the AIR runtime, and the device.


AIR application
ActionScript extension classes
Extension resources (for example, images)
Native implementation
extension
AIR runtime
Device operating system and libraries

Native extension architecture

Native code programming languages

Adobe AIR provides native code APIs that your native code implementation uses for interacting with the ActionScript extension classes. These APIs are available in:

- the C programming language.
- Java

Your native code implementation uses either the C APIs or the Java APIs, but not both, for interacting with the ActionScript extension classes. However, the rest of your native code implementation does not have to exclusively use the same language as the APIs. For example, a developer using the C API can also use:

- C++
- Objective-C
- assembler code to take advantage of highly optimized routines

The following table shows which extension API to use depending on the target device:

Device	Native code API to use
Android devices	Java API with the Android SDK. C API with the Android NDK.
iOS devices	C API
Blackberry PlayBook	C API
Windows desktop devices	C API
Mac OS X desktop devices	C API

Targeting multiple platforms

A native extension often targets multiple platforms. For example, an extension can target devices running iOS and devices running Android. In this case, your ActionScript class implementation and your native code implementation, including the native code language, can vary based on the target platform.

A best practice is for your ActionScript extension classes to provide the same ActionScript public interfaces regardless of their implementation. By keeping the public interfaces the same, you have a true cross-platform native extension. If the ActionScript public interfaces are the same, but the ActionScript implementation is different, you create a different ActionScript library for each platform.

You can also create extensions that do not have a native code implementation for some target platforms. Such an extension is useful in the following situations:

- When only some target platforms support a native implementation of the desired functionality.

  An extension can use a native implementation on those platforms, but use an ActionScript-only implementation on other platforms. For example, consider one platform that provides a specialized mechanism for communication between computer processes. The extension for that platform has a native implementation. The same extension for another platform is ActionScript-only, using ActionScript Socket classes.

  When application developers use the extension, they can write one application without knowing how the extension is implemented on the different target platforms.

- When testing an extension.

  Consider a native extension that uses a specific feature of a mobile device. You can create an ActionScript-only extension for the desktop. Then, an application developer can use the desktop extension for simulation testing during development before testing on the real target device. Similarly, as an extension developer, you can test the ActionScript side of your extension before involving your native code implementation.

When you publish an extension, you specify the target platforms in an extension descriptor file in a `<platform>` element. Each `<platform>` element names a target, such as `iPhone-ARM` or `Windows-x86`. You can also specify a `<platform>` element named `default`. The `default` platform has an ActionScript-only implementation to use on all platforms not specified with a `<platform>` element. For more information, see "Native extension descriptor files" on page 72.

***Note:** The implementation for at least one targeted platform must contain native code. If no targeted platforms require native code, then using native extensions is not the correct choice. In such cases, create a SWC library.*

Extension availability at runtime

A native extension is available at runtime to an application in one of the following ways:

Application-bundling The extension is packaged with the AIR application, and installed with the application onto the target device. An extension package typically contains the native and ActionScript implementations for multiple platforms, but can contain only one platform's native and ActionScript implementations. Sometimes the extension package also contains an ActionScript-only implementation for unsupported platforms or for test platforms.

Device-bundling The extension is installed independently of any AIR application in a directory on the target device. To use device-bundling, you typically work with the device manufacturer to install the extension on the device.

The following table shows which devices support application-bundling and device-bundling:

	Application-bundling	**Device bundling**
Android	Yes	No
iOS	Yes	No
Blackberry PlayBook	Yes	Yes
Windows	Yes	No
Mac OS X	Yes	No

Extension contexts

A native extension is loaded once each time an application runs. However, to use the native implementation, the ActionScript part of your extension calls a special ActionScript API to create an *extension context*.

A native extension can do either of the following.

- Create only one extension context.

  Only one extension context is typical for a simpler extension that provides only one set of functions in the native implementation.

- Create multiple extension contexts that co-exist.

  Multiple extension contexts are useful to associate ActionScript objects with native objects. Each association between an ActionScript object and a native object is one extension context instance. These extension context instances can have different context types. The native implementation can provide a different set of functions for each context type.

Each extension context can have context-specific data that you define and use in your native implementation.

An extension context can only be created by the ActionScript code in an extension. It cannot be created by the native code or by the application code.

Task overview to create a native extension

To create a native extension, do the following tasks:

1. Define the methods and properties of the ActionScript extension classes.
2. Code the ActionScript extension classes.

   See “Coding the ActionScript side” on page 7.

3. Code the native implementation.

   See “Coding the native side with C” on page 15 and “Coding the native side with Java” on page 30.

4. Build the ActionScript side and the native side, create an extension descriptor file, and package the extension and its resources.

   See “Packaging a native extension” on page 39 for all devices.

5. Document the public interfaces of the ActionScript extension class.

Typically, as with any software development, working through these steps is an iterative process.

Chapter 2: Coding the ActionScript side

A native extension is made of two parts:

- ActionScript extension classes you define.
- A native implementation.

The ActionScript extension classes access and exchange data with the native implementation. This access is provided with the ActionScript class ExtensionContext. Only ActionScript code that is part of an extension can access the ExtensionContext class methods.

Coding the ActionScript side of your extension includes the following tasks:

- Declaring the public interfaces of your ActionScript extension class.
- Using the static method `ExtensionContext.createExtensionContext()` to create an ExtensionContext instance.
- Using the `call()` method of the ExtensionContext instance to call methods in the native implementation.
- Adding event listeners to the ExtensionContext instance to listen for events dispatched from the native implementation.
- Using the `dispose()` method to delete the ExtensionContext instance.
- Sharing data between the ActionScript side and the native side. The data shared can be any ActionScript object.
- Using the `getExtensionDirectory()` method to access the directory in which the extension is installed. All information and resources related to the extension are in this directory. (An exception to this rule exists for iOS devices.)

For examples of native extensions, see Native extensions for Adobe AIR.

For more information about the ExtensionContext class, see the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

Declare the public interfaces

The first step in creating a native extension is determining the extension's public interfaces. Application code uses these public interfaces to interact with the extension. ActionScript code goes in files with the .as extension. Create a .as file with your class definition. For example, the following code shows the declaration of a simple TVChannelController extension class, without yet filling in its implementation. This simple class allows an application to manipulate the channel setting on a hypothetical TV.

```
package com.example {
    public class TVChannelController extends EventDispatcher {

        public function TVChannelController() {
        }

        public function set currentChannel(channelToSet:int):void {
        }

        public function get currentChannel():int {
        }
    }
}
```

***Note:** When designing your public interfaces, consider whether you will release subsequent versions of your extension. If so, consider backward compatibility support in your initial design. For more information about backward compatibility issues for device-bundled extensions, see "Native extension backward compatibility" on page 13.*

Check for native extension support

A best practice is to always define a public interface that provides a handshake between the native extension and the AIR application. Instruct AIR application developers using your extension to check this method before calling any other extension method.

For example, consider an ActionScript extension class public interface called `isSupported()`. The `isSupported()` method allows an AIR application to make logic decisions based on whether the device on which the application is running supports the extension. If `isSupported()` returns `false`, the AIR application must decide what to do without the extension. For example, the AIR application can decide to exit.

Create an ExtensionContext instance

To begin working with the native implementation, the ActionScript extension class uses the ExtensionContext static method `createExtensionContext()`. This method returns a new instance of the ExtensionContext class.

```
package com.example {
    public class TVChannelController extends EventDispatcher {

        private var extContext:ExtensionContext;

        public function TVChannelController() {
            extContext = ExtensionContext.createExtensionContext(
                            "com.example.TVControllerExtension", "channel");

        }
        .
        .
        .
    }
}
```

In this example, the constructor calls `createExtensionContext()`. Although your extension classes can call `createExtensionContext()` in any method, typically a constructor or other initialization method calls it. Save the returned ExtensionContext instance in a data member of the class.

Note: *Calling* `createExtensionContext()` *as part of a static data member's definition is not recommended. Doing so means that the runtime creates the extension context earlier than the application needs it. If the application's execution path does not eventually use the extension, creating the context wastes device resources.*

The method `createExtensionContext()` takes two parameters: an extension ID and a context type.

The extension ID

The method `createExtensionContext()` takes a String parameter that is the identifier, or name, of the extension. This name is the same name you use in the extension descriptor file in the `id` element. (You create the extension descriptor file when you package your extension). The application developers also use this name in the `extensionID` element in their application descriptor file. If an extension with the specified name is not available, then `createExtensionContext()` returns `Null`.

To avoid name conflicts, Adobe recommends using reverse DNS for an extension ID. For example, the ID of the TVControllerChannel extension is `com.example.TVControllerExtension`. Because all extensions share a single, global namespace, using reverse DNS for the extension ID avoids name conflicts between extensions.

The context type

The method `createExtensionContext()` takes a String parameter that is the context type for the new extension context. This string specifies more information about what the new extension context is to do.

For example, suppose the extension com.example.TVControllerExtension can manipulate both channel and volume settings. Passing `"channel"` or `"volume"` in `createExtensionContext()` indicates which functionality the new extension context will be used for. Another ActionScript class in the extension, such as TVVolumeController, could call `createExtensionContext()` with `"volume"` for the `contextType` value. The native implementation uses the contextType value in its context initialization.

Typically, each possible context type value you define corresponds to a different set of methods in your native implementation. The context type, therefore, corresponds to what is, in effect, a class in your native implementation. If you call `createExtensionContext()` multiple times with the same context type, typically your native implementation creates multiple instances of a particular native class.

When the context types of multiple calls to `createExtensionContext()` are different, the native side typically performs different initializations. Depending on the context type, the native side can create an instance of a different native class and can provide a different set of native functions.

Note: *A simple extension often has only one context type. That is, it has only one set of methods in the native implementation. In this simple case, the context type String parameter can be* `Null`.

Call a native function

After the ActionScript extension class has called `ExtensionContext.createExtensionContext()`, it can call methods in the native implementation. The TVChannelController example calls native methods `"setDeviceChannel"` and `"getDeviceChannel"` as follows:

```
package com.example {
    public class TVChannelController extends EventDispatcher {

        private var extContext:ExtensionContext;
        private var channel:int;

        public function TVChannelController() {
            extContext = ExtensionContext.createExtensionContext(
                            "com.example.TVControllerExtension", "channel");

        }

        public function set currentChannel(channelToSet:int):void {
            extContext.call("setDeviceChannel", channelToSet);
        }

        public function get currentChannel():int {
            channel = int (extContext.call("getDeviceChannel"));
            return channel;
        }

}
```

The `call()` method of ExtensionContext takes these parameters:

- `functionName`. This string represents a function in the native implementation. In the TVChannelController example, these strings are different from the ActionScript method names. You can choose to make the names the same. You can also choose whether a `functionName` string is the same as the name of the native function that it represents. In your native implementation, you provide the association between this `functionName` string and the native function. The association is in an output parameter of your `FREContextInitializer()` method. See "Extension context initialization" on page 17.
- An optional list of parameters. Each parameter is passed to the native function. A parameter can be a primitive type, such as an int, or any ActionScript Object.

The return value of the `call()` method of ExtensionContext is a primitive type or any ActionScript Object. The subclass of Object that it returns depends on what the native function returns. For example, the native function `"getDeviceChannel"` returns an int.

Listen for events

The native implementation can dispatch events that the ActionScript extension code can listen for. This mechanism allows the native implementation to perform tasks asynchronously, notifying the ActionScript side when the task is complete.

The event target is the ExtensionContext instance. Therefore, use the `addEventListener()` method of the ExtensionContext instance to subscribe to events from the native implementation.

The following example adds code to TVChannelController to receive an event from the native implementation. The application using the extension calls the ActionScript extension class method `scanChannels()`, which in turn calls the native function `"scanDeviceChannels"`.

This native function asynchronously scans for all available channels. When it has completed the scan, it dispatches an event. The `onStatus()` method handles the event by querying the native method `"getDeviceChannels"` for the list of channels. The `onStatus()` method stores the list in the `scannedChannelList` data member, and dispatches an event to the application's listening object. When the application object receives the event, it can call the ActionScript extension class property accessor `availableChannels`.

```
package  com.example {
     public  class TVChannelController extends EventDispatcher {

        private var extContext:ExtensionContext;
        private var channel:int;
        private var scannedChannelList:Vector.<int>;

        public function TVChannelController() {
            extContext = ExtensionContext.createExtensionContext(
                            "com.example.TVControllerExtension", "channel");
            extContext.addEventListener(StatusEvent.STATUS, onStatus);
        }
        .
        .
        .
        public function scanChannels():void {
            extContext.call("scanDeviceChannels");
        }
        public function get availableChannels():Vector.<int> {
            return scannedChannelList;
        }
        private function onStatus(event:StatusEvent):void {
            if ((event.level == "status") && (event.code == "scanCompleted")) {
                scannedChannelList = (Vector.<int>)(extContext.call("getDeviceChannels"));
                dispatchEvent (new Event ("scanCompleted") );
            }
        }
    }
}
```

The example illustrates the following points:

- The native implementation can dispatch only a StatusEvent object. Therefore, the `addEventListener()` method listens for the event type `StatusEvent.STATUS`.
- The native implementation sets the `code` and `level` properties of the StatusEvent object. You can define the strings you want to use for these properties. In this example, the native implementation sets the `level` property to "`status"` and the `code` property to `"scanCompleted"`. Typically, the `level` property of a StatusEvent has the value `"status"`, `"info"`, or `"error"`.
- Because TVChannelController is a subclass of EventDispatcher, it can also dispatch an event. In this example, it dispatches an Event object with the `type` property `"scanCompleted"`. Any ActionScript object interested in this event can listen for it. For example, the following code shows a snippet from an AIR application that uses this extension. The application creates a TVChannelController object. Then, it asks the TVChannelController object to scan for channels. Then it waits for the scan to complete.

```
var channelController:TVChannelController = new TVChannelController();
channelController.addEventListener("scanCompleted", onChannelsScanned);
channelController.scanChannels();
var channelList:Vector.<int>;

private function onChannelsScanned(evt:Event):void {
    if (evt.type == "scanCompleted") {
        channelList = channelController.availableChannels;}
}
```

Dispose of an ExtensionContext instance

The ActionScript extension class can dispose of the ExtensionContext instance by calling the ExtensionContext method `dispose()`. This method notifies the native implementation to clean up resources that the instance uses. For example, the TVChannelController class can add a method for cleaning up:

```
public function dispose (): void {
    extContext.dispose();
    // Clean up other resources that the TVChannelController instance uses.
}
```

Your ActionScript extension class does not have to explicitly call the ExtensionContext instance's `dispose()` method. In this case, the runtime calls it when the runtime garbage collector disposes of the ExtensionContext instance. A best practice, however, is to explicitly call `dispose()`. An explicit call to `dispose()` typically cleans up resources much sooner than waiting for the garbage collector.

Whether called explicitly or by the garbage collector, the ExtensionContext `dispose()` method results in a call to the native implementation's context finalizer. For more information, see "Extension context finalization" on page 18.

Access the native extension's directory

Sometimes extensions include additional files, such as images. An extension sometimes also wants to access the information in the extension descriptor file, such as the extension version number.

To access these files for extensions on all devices except iOS devices, use the ExtensionContext class static method `getExtensionDirectory()`. For example:

```
var extDir:File =
ExtensionContext.getExtensionDirectory("com.example.TVControllerExtension");
```

Pass the name of the extension to `getExtensionDirectory()`. This String value is the same name you use in:

- the extension descriptor file in the `id` element.
- the extension ID parameter you pass to `ExtensionContext.createExtensionContext()`.

The returned File instance refers to the base extension directory. The extension directory has the following structure:

```
extension base directory/
    platform independent files
    META-INF/
        ANE/
            extension.xml
            platform name/
                platform dependent files and directories
```

Regardless where the extension directory is on the device, the extension's files are always in the same location relative to the base extension directory. Therefore, use the returned File instance and File class methods to navigate to and manipulate specific files included with the extension.

The extension directory location depends on whether the extension is available through application-bundling or device-bundling as follows:

- With application-bundling, the extension directory is located within the application directory.
- With device-bundling, the extension directory location depends on the device.

An exception to using `getExtensionDirectory()` exists for ActionScript extensions for iOS devices. The resources for these extensions are not located in the extension directory. Instead, they are located in the top-level application directory. For more information, see "Resources on iOS devices" on page 54.

More Help topics

"Extension availability at runtime" on page 5

Identify the calling application from a native extension

The ActionScript side of your extension can identify and evaluate the AIR application using the extension. For example, use the ActionScript class NativeApplication to get information about the AIR application, such as its ID and signature data. Then the ActionScript side can make runtime decisions based on this information.

Sometimes the native implementation has similar runtime decisions to make. In this case, the ActionScript side can use the `call()` method of an ExtensionContext instance to report the application information to the native implementation.

Native extension backward compatibility

Backward compatibility and the extension's public interfaces

A best practice is to maintain backward compatibility in your extension's ActionScript public interfaces. Continue to support the extension's classes, methods, properties, and events in all subsequent versions of the extension.

Device-bundled extensions have a more complex issue with regard to backward compatibility. Sometimes, the *behavior* of an extension is different between versions of an extension. For example, a particular method returns a value with a new meaning in a new version of the extension. When this behavior occurs for device-bundled extensions, an application can stop working correctly. This problem can occur if the application was built with a version of the extension that behaves differently than the version of the extension installed on the device. In this case, the application expects one behavior, but the installed extension provides a different behavior.

In such cases, the extension installed on the device can determine how to proceed. The extension can do the following:

- Look up the extension version that the AIR application was built with as well as the version installed on the device.
- Determine whether the extension's behavior is different in the two versions.
- If the AIR application was built with an older version of the extension, revert to the older version's behavior.

  Note: *Typically an AIR application that was built with a newer version of an extension is not available on the device. For more information, see "Backward compatibility and the device's application store" on page 14.*

To look up the extension version number that the application was built with, do the following:

1. Get the application installation directory using `File.applicationDirectory`.
2. Use the File class APIs to access the extension.xml file of the extension that the application built against. The file is at:

   `<application directory>/META-INF/AIR/extensions/<extensionID>/META-INF/ANE/extension.xml`
3. Read the contents of the extension.xml file and find the value of the `<versionNumber>` element.

To look up the installed extension's version number, do the following:

1. Use the static method `ExtensionContext.getExtensionDirectory()` to get the base directory for the extension.
2. Use the File class APIs to access the extension.xml file of the extension installed on the device. The file is at:

   `<extension base directory>/META-INF/ANE/extension.xml`
3. Read the contents of the extension.xml file and find the value of the `<versionNumber>` element.

Backward compatibility and the device's application store

An AIR application that was built with a newer version of the extension than is installed on the device is typically not available on the device. The application is not available because of how device manufacturers handle requests from the device's application store to a server to download such an application. Adobe recommends the following handling to the device manufacturers:

- Consider the case when the server downloads an application that uses a newer version of the extension. The server also downloads the newer version of the extension. The device's application store installs both the application and the newer version of the extension.
- Consider the case when the server cannot download a newer version of the extension. The server also does not download the application that uses that version of the extension. The device's application store handles the scenario gracefully, informing the end user as needed.
- Consider the case when the server downloads an application that uses a newer version of the extension, but does not download the newer version of the extension. The device's application store does not allow the end user to run the application. The application store handles the scenario gracefully, informing the end user as needed.

Chapter 3: Coding the native side with C

Some devices use the C programming language in their native implementations. If you are targeting your native extension for such a device, use the native extensions C API to code the native side of your extension.

The C API is in the file FlashRuntimeExtensions.h. The file is available in the AIR SDK in the `include` directory. The AIR SDK is available at http://www.adobe.com/products/air/sdk/.

The AIR runtime connects the ActionScript side of an extension to the native side of the extension.

Using the C API, you do the following tasks:

- Initialize the extension.
- Initialize each extension context when it is created.
- Define functions that the ActionScript side can call.
- Dispatch events to the ActionScript side.
- Access data passed from the ActionScript side, and pass data back to the ActionScript side.
- Create and access context-specific native data and context-specific ActionScript data.
- Clean up extension resources when the extension's work is done.

For details about each C API function, such as parameters and return values, see "Native C API Reference" on page 81.

For examples of native extensions that use the C API, see Native extensions for Adobe AIR.

Extension initialization

The runtime calls an extension initialization function on the native side. The runtime calls this initialization function once each time the application that uses the extension runs. Specifically, the runtime calls the initialization function the first time the extension calls `ExtensionContext.createExtensionContext()` for any context.

The function initializes data that all extension contexts can use. Define your extension initializer function with the signature of FREInitializer().

For example:

```
void MyExtensionInitializer
    (void** extDataToSet, FREContextInitializer* ctxInitializerToSet,
     FREContextFinalizer* ctxFinalizerToSet)
{
    extDataToSet = NULL;  // This example does not use any extension data.
    *ctxInitializerToSet = &MyContextInitializer;
    *ctxFinalizerToSet = &MyContextFinalizer;
}
```

The `FREInitializer()` method that you define returns the following data to the runtime:

- A pointer to the data that the runtime later passes to each new extension context. For example, if all extension contexts use the same utility library, this data can include a pointer to the library. This data is called the *extension data*.

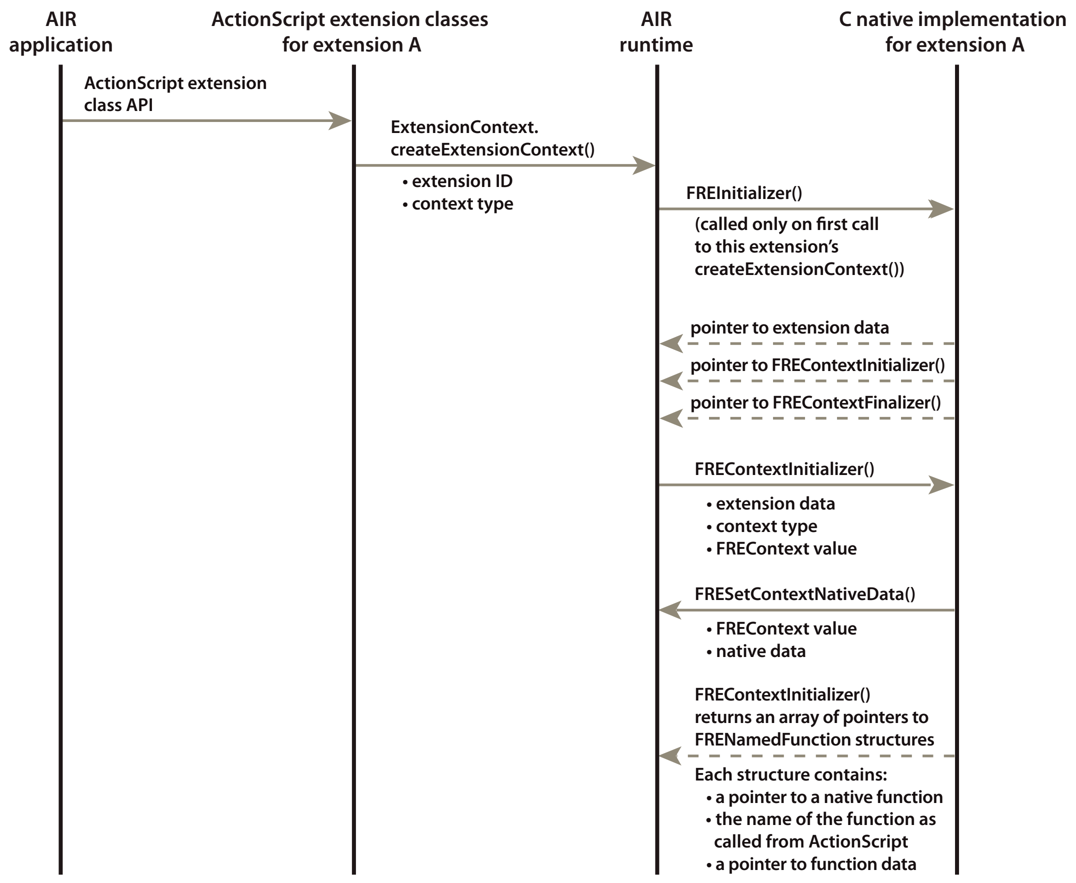
  The extension data can be any data you choose. It can be a simple primitive data type, or a pointer to a structure you define. In this example, the pointer is `NULL` because the extension does not have a use for this data.

- A pointer to the context initialization function. Each time the ActionScript side calls `ExtensionContext.createExtensionContext()`, the runtime calls an extension context initialization function that you provide. See "FREContextInitializer()" on page 88.
- A pointer to the context finalizer function. The runtime calls this function when the runtime disposes of the extension context. This call occurs when the ActionScript side calls the ExtensionContext instance's `dispose()` method. If `dispose()` is not called, the runtime garbage collects the ExtensionContext instance. See "FREContextFinalizer()" on page 87.

For application-bundled extensions, your implementation of `FREInitializer()` can have any name. Specify the name of the initialization function in the extension descriptor file. See "Native extension descriptor files" on page 72.

For device-bundled applications, how to specify the extension initializer function is device-dependent.

The following sequence diagram shows the AIR runtime calling the FREInitializer() function. It also shows context initialization. For more information, see "Extension context initialization" on page 17.


AIR application
ActionScript extension classes for extension A
AIR runtime
C native implementation for extension A
ActionScript extension class API
ExtensionContext. createExtensionContext()
• extension ID
• context type
FREInitializer()
(called only on first call to this extension's createExtensionContext())
pointer to extension data
pointer to FREContextInitializer()
pointer to FREContextFinalizer()
FREContextInitializer()
• extension data
• context type
• FREContext value
FRESetContextNativeData()
• FREContext value
• native data
FREContextInitializer() returns an array of pointers to FRENamedFunction structures
Each structure contains:
• a pointer to a native function
• the name of the function as called from ActionScript
• a pointer to function data

Extension initialization sequence

Extension context initialization

To use the native C methods, the ActionScript side of your extension first calls the static method `ExtensionContext.createExtensionContext()`. Calling `createExtensionContext()` causes the runtime to do the following:

- Create an ExtensionContext instance.
- Create internal data it uses to track the extension context.
- Call the extension context initialization function.

The extension context initialization function initializes the native implementation for the new extension context. Define your extension context initializer function with the signature of FREContextInitializer().

For example, the Vibration example uses the following function:

```
void ContextInitializer(void* extData, const uint8_t* ctxType, FREContext ctx,
                        uint32_t* numFunctionsToSet,
                        const FRENamedFunction** functionsToSet) {

    *numFunctionsToSet = 2;

    FRENamedFunction* func = (FRENamedFunction*)malloc(sizeof(FRENamedFunction)*2);
    func[0].name = (const uint8_t*)"isSupported";
    func[0].functionData = NULL;
    func[0].function = &IsSupported;

    func[1].name = (const uint8_t*)"vibrateDevice";
    func[1].functionData = NULL;
    func[1].function = &VibrateDevice;

    *functionsToSet = func;
}
```

A context initialization function receives the following input parameters:

- The extension data that the extension initialization function had created. See “Extension initialization” on page 15.
- The context type. The ActionScript method `ExtensionContext.createExtensionContext()` is passed a parameter that specifies the context type. The runtime passes this string value to the context initialization function. The function then uses the context type to choose the set of methods in the native implementation that the ActionScript side can call. Each context type typically corresponds to a different set of methods. See “The context type” on page 9.

  The value of the context type is any string agreed to between the ActionScript side and the native side.

  If your extension has only one set of methods in the native implementation, pass null or an empty string in `ExtensionContext.createExtensionContext()`. Then ignore the context type parameter in the extension context initializer.
- A FREContext value. The runtime creates internal data when it creates an extension context. It associates the internal data with the ExtensionContext class instance on the ActionScript side.

  When your native implementation dispatches an event the ActionScript side, it specifies this FREContext value. The runtime uses the FREContext value to dispatch the event to the corresponding ExtensionContext instance. See “FREDispatchStatusEventAsync()” on page 95.

Also, native functions can use this value to access and set the context-specific native data and context-specific ActionScript data.

The extension context initialization function sets the following output parameters:

- An array of native functions. The ActionScript side can call each of these functions by using the ExtensionContext instance's `call()` method.

  The type of each array element is `FRENamedFunction`. This structure includes a string which is the name the ActionScript side uses to call the function. The structure also includes a pointer to the C function you write. The runtime associates the name with the C function. Although the name string does not have to match the actual function name, typically you use the same name.

- The number of functions in the array of native functions.

A sequence diagram showing the AIR runtime calling the `FREContextInitializer()` function is in "Extension initialization" on page 15.

More Help topics

Vibration native extension example

Context-specfic data

Context-specific data is specific to an extension context. (Recall that extension data is for all extension contexts in an extension). The context initialization method, context finalization method, and native extension methods can create, access, and modify the context-specific data.

The context-specific data can include the following:

- Native data. This data is any data you choose. It can be a simple primitive data type, or a structure you define. See "FREGetContextNativeData()" on page 99 and "FRESetContextNativeData()" on page 114.
- ActionScript data. This data is an FREObject variable. Since an FREObject variable corresponds to an ActionScript class object, this data allows you to save and later access an ActionScript object. See "FREGetContextActionScriptData()" on page 98 and "FRESetContextActionScriptData()" on page 113. Also see "The FREObject type" on page 22.

A sequence diagram showing the native implementation setting context-specific native data is in "Extension initialization" on page 15. A sequence diagram showing the native implementation getting the context-specific data is in "Extension functions" on page 20.

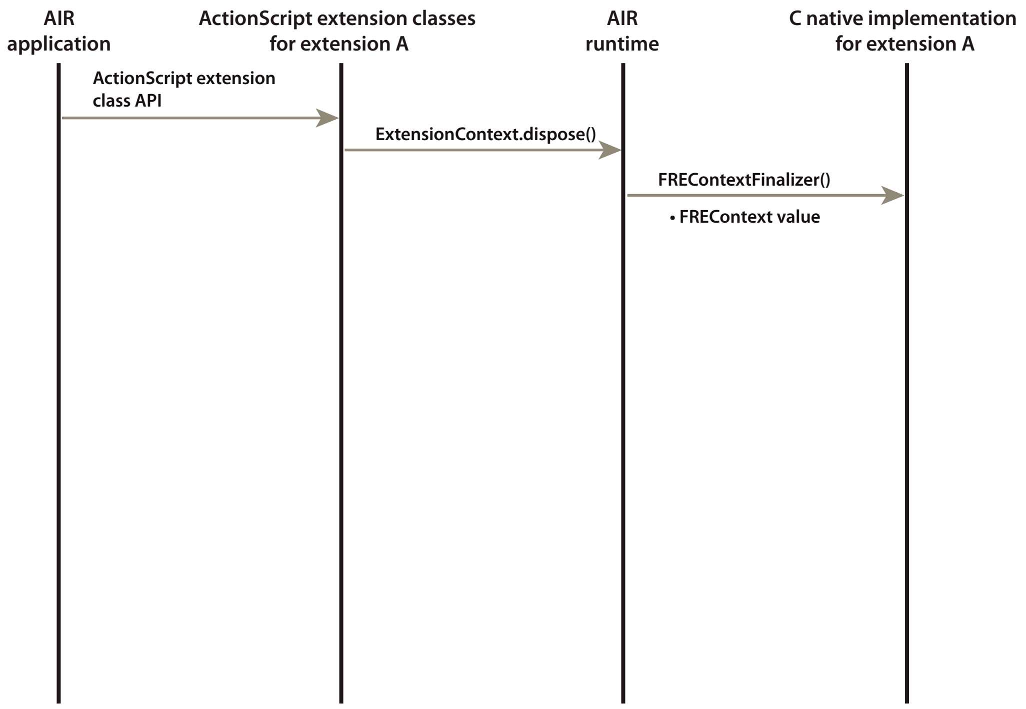
Extension context finalization

The ActionScript side of your extension can call the `dispose()` method of an ExtensionContext instance. Calling `dispose()` causes the runtime to call the context finalization function of your extension. Define your extension context finalization function with the signature of FREContextFinalizer().

This method has one input parameter: the FREContext value. You can pass this FREContext value to `FREGetContextNativeData()` and `FREGetContextActionScriptData()` to access the context-specific data. Clean up any data and resources associated with this context.

If the ActionScript side does not call `dispose()`, the runtime garbage collector disposes of the ExtensionContext instance when no more references to it exist. At that time, the runtime calls your context finalization function.

The following sequence diagram shows the AIR runtime calling the `FREContextFinalizer()` function:


AIR application
ActionScript extension classes for extension A
AIR runtime
C native implementation for extension A
ActionScript extension class API
ExtensionContext.dispose()
FREContextFinalizer()
• FREContext value

Extension context finalization sequence

Extension finalization

The C API provides an extension finalization function for the runtime to call when it unloads the extension. However, the runtime does not always unload an extension. Therefore, the runtime does not always call the extension finalization function.

Define your extension finalization function with the signature of FREFinalizer(). This method has one input parameter: the extension data you created in your extension initialization function. Clean up any data and resources associated with this extension.

For application-bundled extensions, your implementation of `FREFinalizer()` can have any name. Specify the name of the finalization function in the extension descriptor file. See “Native extension descriptor files” on page 72.

For device-bundled applications, how to specify the extension finalization function is device-dependent.

Extension functions

The ActionScript side of your extension calls C functions you implement by calling the ExtensionContext instance's `call()` method. The `call()` method takes these parameters:

- The name of the function. You provided this name in an output parameter of your context initialization function. This name is an arbitrary string agreed to between the ActionScript side and the native side. Typically, it is the same name as the actual name of the native C function. However, these names can be different because the runtime associates the arbitrary name with the actual function.
- A list of arguments for the native function. These arguments can be any ActionScript objects: primitive types or ActionScript class objects.

Define each of your native functions with the same function signature: FREFunction(). The runtime passes the following parameters to each native function:

- The FREContext value. The native function can use this value to access and set the context-specific data. Also, the native implementation uses the FREContext value to dispatch an asynchronous event back to the ActionScript side.
- A pointer to the data associated with the function. This data is any native data. When the runtime calls the native function, it passes the function this data pointer.
- The number of function parameters.
- The function parameters. Each function parameter has the type FREObject. These parameters correspond to ActionScript class objects or primitive data types.

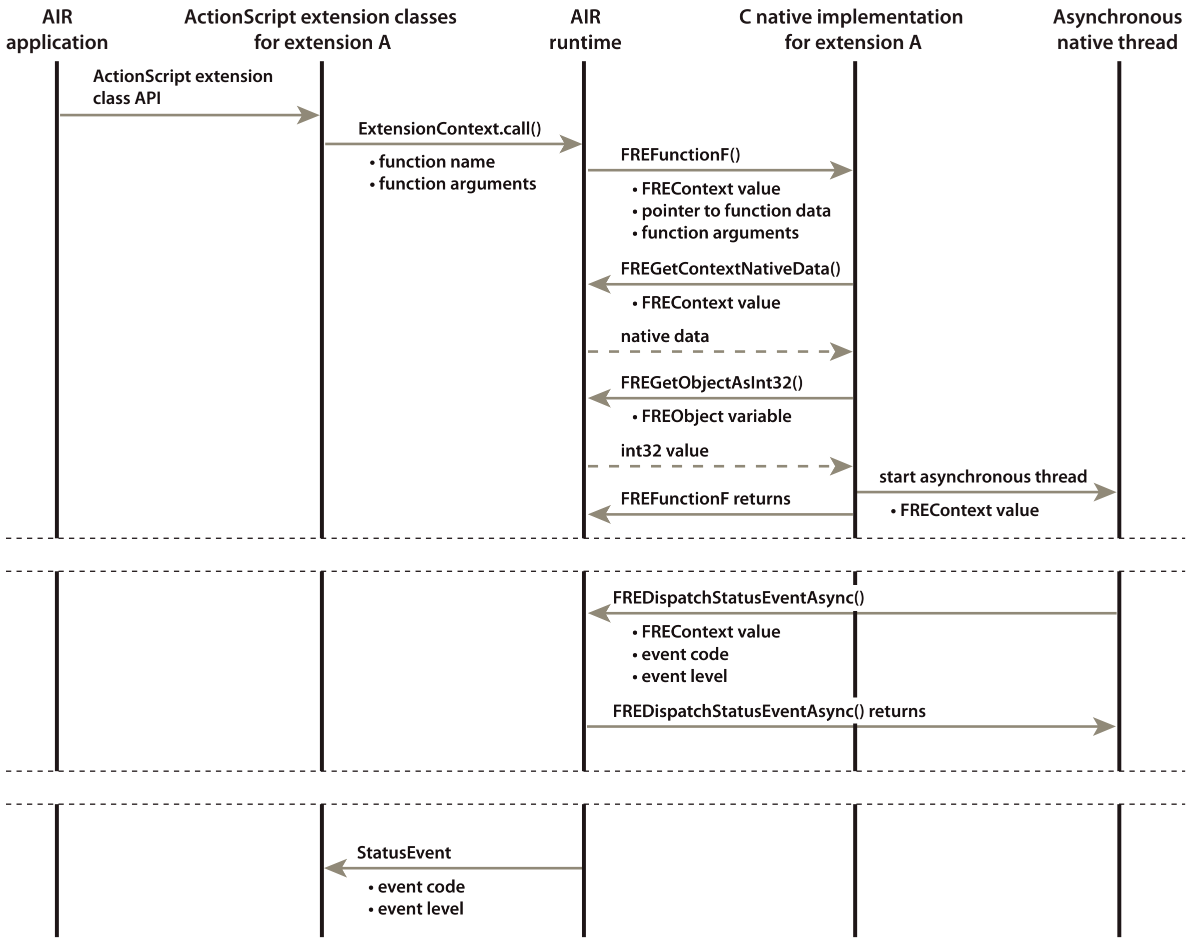
A native function also has a return value with the type FREObject. The runtime returns the corresponding ActionScript object as the return value for the ExtensionContext `call()` method.

***Note:** Do not set a native function's visibility to hidden. Use the default visibility.*

The following sequence diagram shows an AIR application making a function call that results in calling a native C function named `FREFunctionF()`. In this example, the C function:

- Gets the context-specific native data.
- Gets the int32 value of an ActionScript object.
- Starts an asynchronous thread which later dispatches an event.

***Note:** The behavior of the C function `FREFunctionF()` is only a sample behavior to illustrate a call sequence.*



Native function sample call sequence

Dispatching asynchronous events

The native C code can dispatch asynchronous events back to the ActionScript side of your extension. For example, an extension method can start another thread to perform some task. When the task in the other thread completes, that thread calls `FREDispatchStatusEventAsync()` to inform the ActionScript side of the extension. The target of the event is an ActionScript ExtensionContext instance.

The sequence diagram in "Extension functions" on page 20 shows a native C function starting an asynchronous thread, which later dispatches an event.

More Help topics

"FREDispatchStatusEventAsync()" on page 95

The FREObject type

A variable of type FREObject refers to an object that corresponds to an ActionScript class object or primitive type. You use an FREObject variable in your native implementation to work with ActionScript data. A primary use of the FREObject type is for native function parameters and return values.

When you write a native function, you decide on the order of the parameters. Since you also write the ActionScript side, you use that parameter order in the ExtensionContext instance's `call()` method. Therefore, although every native function parameter is an FREObject variable, you know its corresponding ActionScript type.

Similarly you decide on the ActionScript type of the return value, if any, of a native function. The `call()` method returns an object of this type. Although the native function return value is always an FREObject variable, you know its corresponding ActionScript type.

The extensions C API provides functions for using the object that an FREObject variable refers to. Because these objects correspond to ActionScript data, these C API functions are how you access an ActionScript class object or primitive data variable. The C APIs that you use depend on the type of the ActionScript object. The types are the following:

- An ActionScript primitive data type
- An ActionScript class object
- An ActionScript String object
- An ActionScript Array or Vector class object
- An ActionScript ByteArray class object
- An ActionScript BitmapData class object

***Note:** You can call the extensions C APIs only from the same thread as the one in which the FREFunction function is running. The one exception is the C API for dispatching an event to the ActionScript side. You can call that function,* `FREDispatchStatusEventAsync()`*, from any thread.*

Determining the type of an FREObject variable

Sometimes you don't know the type of ActionScript Object that an FREObject variable corresponds to. To determine the type, use the C API function FREGetObjectType():

```
FREResult FREGetObjectType( FREObject object, FREObjectType *objectType );
```

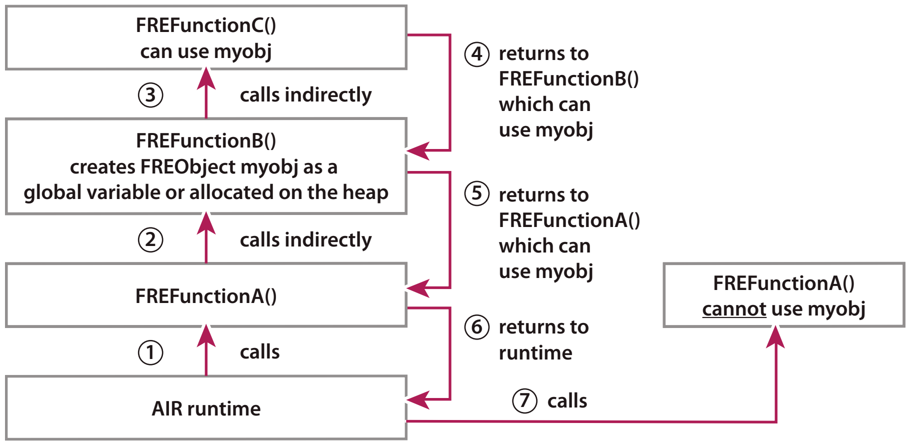
Once you know the type, use the appropriate C APIs to work with the value. For example if the type is `FRE_TYPE_VECTOR`, use the C APIs in "Working with ActionScript Array and Vector objects" on page 27 to work with the Vector object.

FREObject validity

If you attempt to use an invalid FREObject variable in a C API call, the C API returns an `FRE_INVALID_OBJECT` return value.

Any FREObject variable is valid only until the first FREFunction function on the call stack returns. The first FREFunction function on the call stack function is the one that the runtime calls due to the ActionScript side calling the ExtensionContext instance's `call()` method.

The following illustration illustrates this behavior:



FREObject validity on the call stack

***Note:** An FREFunction function can indirectly call another FREFunction function. For example, `FREFunctionA()` can call a method of an ActionScript object. That method then can call `FREFunctionB()`.*

Therefore, when using an FREObject variable, consider the following:

- Any FREObject variable passed to an FREFunction function is valid only until the first FREFunction function on the call stack returns.
- Any FREObject variable that any native function creates using the extensions C API is valid only until the first FREFunction function on the call stack returns.
- You cannot use an FREObject variable in another thread. Only use the FREObject variable in the same thread as the native function that received or created the variable.
- You cannot save an FREObject variable, for example in global data, between calls to FREFunction functions. Because the variable becomes invalid when the first FREFunction function on the call stack returns, the saved variable is useless. However, you can save the corresponding ActionScript object by using the method FRESetContextActionScriptData().
- After an FREObject variable becomes invalid, the corresponding ActionScript object can still exist. For example, if an FREObject variable is a return value of an FREFunction function, its corresponding ActionScript object is still referenced. However, once the ActionScript side deletes its references, the runtime disposes of the ActionScript object.
- You cannot share FREObject variables between extensions.

  ***Note:** You can share FREObject variables between extension contexts of the same extension. However, as in any case, the FREObject variable becomes invalid when the first FREFunction function on the call stack returns to the runtime.*

Working with ActionScript primitive types and objects

Working with ActionScript primitive types

In your native functions, an input parameter can correspond to a primitive ActionScript type. All native function parameters are of type FREObject. Therefore, to work with an ActionScript primitive type input parameter, you get the ActionScript value of the FREObject parameter. You store the value in a corresponding primitive C data type variable. Use the following C API functions:

- FREGetObjectAsInt32()

  ```
  FREResult FREGetObjectAsInt32(FREObject object, int32_t *value);
  ```

- FREGetObjectAsUint32()

  ```
  FREResult FREGetObjectAsUint32(FREObject object, uint32_t *value);
  ```

- FREGetObjectAsDouble()

  ```
  FREResult FREGetObjectAsDouble(FREObject object, double *value);
  ```

- FREGetObjectAsBool(),

  ```
  FREResult FREGetObjectAsBool  (FREObject object, bool *value);
  ```

If an output parameter or return value corresponds to a primitive ActionScript type, you create the ActionScript primitive using a C API function. You provide a pointer to an FREObject variable and the value of the primitive in a C data variable. The runtime creates the ActionScript primitive and sets the FREObject variable to correspond to it. Use the following C API functions:

- FRENewObjectFromInt32()

  ```
  FREResult FRENewObjectFromInt32(int32_t value, FREObject *object);
  ```

- FRENewObjectFromUint32()

  ```
  FREResult FRENewObjectFromUint32(uint32_t value, FREObject *object);
  ```

- FRENewObjectFromDouble()

  ```
  FREResult FRENewObjectFromDouble(double value, FREObject *object);
  ```

- FRENewObjectFromBool(),

  ```
  FREResult FRENewObjectFromBool  (bool value, FREObject *object);
  ```

Working with ActionScript String objects

In your native functions, an input parameter can correspond to an ActionScript String class object. All native function parameters are of type FREObject. Therefore, to work with an ActionScript String parameter, you get the ActionScript String value of the FREObject parameter. You store the value in a corresponding C string variable. Use the C API function FREGetObjectAsUTF8():

```
FREResult FREGetObjectAsUTF8(
            FREObject       object,
            uint32_t*       length,
            const uint8_t** value
);
```

After calling `FREGetObjectAsUTF8()`, the ActionScript String value is in the `value` parameter, and the `length` parameter tells the length of the value string in bytes.

If an output parameter or return value corresponds to an ActionScript String class object, you create the ActionScript String object using a C API. You provide a pointer to a FREObject variable and the string value and length in bytes in C string variables. The runtime creates the ActionScript String object and sets the FREObject variable to correspond to it. Use the C API function FRENewObjectFromUTF8():

```
FREResult FRENewObjectFromUTF8(
            uint32_t        length,
            const uint8_t*  value,
            FREObject*      object
);
```

The `value` parameter strings must use UTF- 8 encoding and include the null terminator.

***Note:** All string parameters to any C API function use UTF-8 encoding and include the null terminator.*

Working with ActionScript Class objects

In your native functions, an input parameter can correspond to an ActionScript class object. Since all native function parameters are of type FREObject, the C APIs provide functions for manipulating class objects using an FREObject variable.

Use the following C API functions to get and set a property of the ActionScript class object:

- FREGetObjectProperty()

  ```
  FREResult FREGetObjectProperty(
              FREObject object,
              const uint8_t*  propertyName,
              FREObject*       propertyValue,
              FREObject*      thrownException
  );
  ```

- FRESetObjectProperty()

  ```
  REResult FRESetObjectProperty(
              FREObject       object,
              const uint8_t*  propertyName,
              FREObject       propertyValue,
              FREObject*      thrownException
  );
  ```

Use the following C API to call a method of an ActionScript class object:

FRECallObjectMethod()

```
FREResult FRECallObjectMethod(
            FREObject       object,
            const uint8_t*  methodName,
            uint32_t        argc,
            FREObject       argv[],
            FREObject*      result,
            FREObject*      thrownException
);
```

If an output parameter or return value corresponds to an ActionScript class object, you create the ActionScript object using a C API. You provide a pointer to an FREObject variable plus FREObject variables to correspond to parameters to the ActionScript class constructor. The runtime creates the ActionScript class object and sets the FREObject variable to correspond to it. Use the following C API function:

FRENewObject()

```
FREResult FRENewObject(
        const uint8_t* className,
        uint32_t       argc,
        FREObject      argv[],
        FREObject*     object,
        FREObject*     thrownException
);
```

***Note:** These general ActionScript object manipulation functions apply to all ActionScript class objects. However, the ActionScript classes Array, Vector, ByteArray, and BitmapData are special cases because they each involve large amounts of data. Therefore, the C API provides additional specific functions for manipulating objects of these special cases.*

Working with ActionScript ByteArray objects

Use the ActionScript ByteArray class to efficiently pass many bytes between the ActionScript side and native side of your extension. In your native functions, an input parameter, output parameter, or return value can correspond to an ActionScript ByteArray class object.

As with other ActionScript class objects, an FREObject variable is the native side representation of an ActionScript ByteArray object. The C APIs provide functions for manipulating a ByteArray class object using an FREObject variable. Use `FRESetObjectProperty()`, `FREGetObjectProperty()`, and `FRECallObjectMethod()` to get and set the ActionScript ByteArray object's properties and to call its methods.

However, to manipulate the bytes of the ByteArray object in the native code, use the C API function FREAcquireByteArray(). This method accesses the bytes of a ByteArray object that was created on the ActionScript side:

```
FREResult FREAcquireByteArray(
            FREObject     object,
            FREByteArray* byteArrayToSet
);
// The type FREByteArray is defined as:

typedef struct {
            uint32_t length;
            uint8_t* bytes;
} FREByteArray;
```

After you have manipulated the bytes, use the C API FREReleaseByteArray():

```
FREResult FREReleaseByteArray( FREObject object );
```

***Note:** Do not call any C API functions between the calls to `FREAcquireByteArray()` and `FREReleaseByteArray()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the byte array contents.*

Example

This example shows the ActionScript side of the extension creating a ByteArray object and initializing its bytes. Then it calls a native function to manipulate the bytes.

```
// ActionScript side of the extension

var myByteArray:ByteArray = new ByteArray();
myByteArray.writeUTFBytes("Hello, World");
myByteArray.position = 0;
myExtensionContext.call("MyNativeFunction", myByteArray);


// C code
FREObject MyNativeFunction(FREContext ctx, void* funcData, uint32_t argc, FREObject argv[]) {

    FREByteArray byteArray;
    int retVal;

    retVal = FREAcquireByteArray(argv[0], &byteArray);
    uint8_t* nativeString = (uint8_t*) "Hello from C";
    memcpy (byteArray.bytes, nativeString, 12);
    retVal = FREReleaseByteArray(argv[0]);

    return NULL;
}
```

Working with ActionScript Array and Vector objects

Use the ActionScript Vector and Array classes to efficiently pass arrays between the ActionScript side and native side of your extension. In your native functions, an input parameter, output parameter, or return value can correspond to an ActionScript Array or Vector class object.

As with other ActionScript class objects, an FREObject variable is the native side representation of an ActionScript Array or Vector object. The C APIs provide functions for manipulating an Array or Vector class object using an FREObject variable.

Use the following C API functions to get and set the length of an Array or Vector object:

- FREGetArrayLength()

  ```
  FREResult FREGetArrayLength(
              FREObject  arrayOrVector,
              uint32_t*  length
  );
  ```

- FRESetArrayLength()

  ```
  FREResult FRESetArrayLength(
              FREObject  arrayOrVector,
              uint32_t   length
  );
  ```

Use the following C API functions to get and set an element of an Array or Vector object:

- FREGetArrayElementAt()

  ```
  FREResult FREGetArrayElementAt(
              FREObject  arrayOrVector,
              uint32_t   index,
              FREObject* value
  );
  ```

- FRESetArrayElementAt()

```
FREResult FRESetArrayElementAt(
        FREObject  arrayOrVector,
        uint32_t   index,
        FREObject  value
);
```

Working with ActionScript BitmapData objects

Use the ActionScript BitmapData class to pass bitmaps between the ActionScript side and native side of your extension. In your native functions, an input parameter, output parameter, or return value can correspond to an ActionScript BitmapData class object.

As with other ActionScript class objects, an FREObject variable is the native side representation of an ActionScript BitmapData object. The C APIs provide functions for manipulating a BitmapData class object using an FREObject variable. Use `FRESetObjectProperty()`, `FREGetObjectProperty()`, and `FRECallObjectMethod()` to get and set the ActionScript ByteArray object's properties and to call its methods.

However, to manipulate the bits of the BitmapData object in the native code, use the C API function FREAcquireBitmapData() or FREAcquireBitmapData2(). These methods access the bits of a BitmapData object that was created on the ActionScript side:

```
FREResult FREAcquireBitmapData(
            FREObject       object,
            FREBitmapData*  descriptorToSet
);
// The type FREBitmapData is defined as:

typedef struct {
    uint32_t    width;
    uint32_t    height;
    bool        hasAlpha;
    bool        isPremultiplied;
    uint32_t    lineStride32;
    uint32_t*   bits32;
} FREBitmapData;

//Or:
FREResult FREAcquireBitmapData2(
            FREObject       object,
            FREBitmapData2* descriptorToSet
);
// The type FREBitmapData is defined as:

typedef struct {
    uint32_t    width;
    uint32_t    height;
    bool        hasAlpha;
    bool        isInvertedY;
    bool        isPremultiplied;
    uint32_t    lineStride32;
    uint32_t*   bits32;
} FREBitmapData2;
```

All the fields of a `FREBitmapData` or `FREBitmapData2` structure are read-only. However, the `bits32` field points to the actual bitmap values, which you can modify in your native implementation. The `FREBitmapData2` structure and `FREAquireBitmapData2` function were added to the API in AIR 3.1. `FREBitmapData2` contains one additional field, `isInvertedY`, which indicates the order in which the rows of image data are stored.

To indicate that the native implementation has modified all or part of the bitmap, invalidate a rectangle of the bitmap. Use the C API function FREInvalidateBitmapDataRect():

```
FREResult FREInvalidateBitmapDataRect(
        FREObject object,
        uint32_t x,
        uint32_t y,
        uint32_t width,
        uint32_t height
);
```

The `x` and `y` fields are the coordinates of the rectangle to invalidate, relative to the 0,0 coordinates which are the top, left corner of the bitmap. The `width` and `height` fields are the dimensions in pixels of the rectangle to invalidate.

After you have manipulated the bitmap, use the C API function FREReleaseBitmapData():

```
FREResult FREReleaseBitmapData(FREObject object);
```

Note: *Do not call any C API function except `FREInvalidateBitmapDataRect()` between the calls to `FREAcquireBitmapData()` and `FREReleaseBitmapData()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the bitmap contents.*

Threads and native extensions

When coding your native implementation, consider the following:

- The runtime can concurrently call an extension context's FREFunction functions on different threads.
- The runtime can concurrently call different extension contexts' FREFunction functions on different threads.

Therefore, code your native implementations appropriately. For example, if you use global data, protect it with some form of locks.

Note: *Your native implementation can choose to create separate threads. If it does, consider the restrictions specified in "FREObject validity" on page 22.*

Chapter 4: Coding the native side with Java

Android devices use Java as their primary application development language. If you are targeting your native extension for such a device, use the native extensions Java API to code the "native" side of your extension. You can also use the AIR extension C API with the Android Native Development Kit for some purposes. For information on using the C API, see "Coding the native side with C" on page 15.

The Java API is provided in the file FlashRuntimeExtensions.jar. The file is available in the AIR SDK in the `lib/android` directory. The AIR SDK is available at http://www.adobe.com/products/air/sdk/.

To code the Java side of a native extension for Adobe AIR, do the following:

- Implement the FREExtension interface.
- Extend the FREContext abstract class with one or more concrete subclasses.
- Implement the FREFunction interface for each Java function that can be called from the ActionScript side of the extension.

For details about each Java API function, such as parameters and return values, see "Android Java API Reference" on page 116.

Implementing the FREExtension interface

Every extension using the Java API must implement the FREExtension interface. This FREExtension instance is the initial entry point for the Java code in your extension. Specify the fully qualified name of the class in an `<initializer>` element in the extension descriptor file. Java implementations can be used for the Android-ARM platform only. See "Native extension descriptor files" on page 72.

The `createContext()` method is the most important part of the FREExtension implementation. The AIR runtime calls your `createContext()` method when the ActionScript code of the extension calls `ExtensionContext.createExtensionContext()`. The method must return an instance of the FREContext class. The ActionScript `createExtensionContext()` method has a string parameter which is passed to the Java `createContext()` function. You can use this value to provide different contexts for different purposes. If your extension only uses a single context class, you can ignore the parameter.

The other methods of the FREExtension interface, `initialize()` and `dispose()`, are called by the runtime automatically and can be used to create and clean up any persistent resources needed by the extension. However, not every extension needs to do anything in these functions.

The constructor for your FREExtension implementation class must not take any parameters.

The AIR runtime instantiates your FREExtension instance the first time your ActionScript code calls `createExtensionContext()`. The sequence of calls to your Java extension class is:

- FREExtension implementation class constructor
- initialize()
- createContext()

FREExtension example

The following example illustrates a simple FREExtension implementation. This example uses a single extension context. A reference to the context is created the first time the AIR runtime calls the createContext() method. That reference is saved for subsequent use.

```
package com.example;

import android.util.Log;

import com.adobe.fre.FREContext;
import com.adobe.fre.FREExtension;

public class Extension implements FREExtension {
    private static final String EXT_NAME = "Extension";
    private ExtensionContext context;
    private String tag = EXT_NAME + "ExtensionClass";

    public FREContext createContext(String arg0) {
        Log.i(tag, "Creating context");
        if( context == null) context = new ExtensionContext(EXT_NAME);
        return context;
    }

    public void dispose() {
        Log.i(tag, "Disposing extension");
        // nothing to dispose for this example
    }

    public void initialize() {
        Log.i(tag, "Initialize");
        // nothing to initialize for this example
    }
}
```

ExtensionContext example

The following example illustrates the ActionScript code in an extension that exposes the native extension functionality to application code.

```
package com.example
{
import flash.external.ExtensionContext;

public class ExampleExtension
{
    private const extensionID:String = "com.example.Extension";
    private var extensionContext:ExtensionContext;

    public function ExampleExtension()
    {
        extensionContext = ExtensionContext.createExtensionContext( extensionID, null );
    }

    public function usefulFunction( value:Boolean ):Boolean
    {
        var retValue:Boolean = false;
        retValue = extensionContext.call( "usefulFunctionKey", value );
        return retValue;
    }
}
}
```

Application code example

To use a native extension, an application accesses the ActionScript classes and methods provided by the extension. (In the same way it would access the classes and methods of any other ActionScript library.)

```
var exampleExtension:ExampleExtension = new ExampleExtension();

var input:Boolean = true;
var untrue:Boolean = exampleExtension.usefulFunction( input );
```

Extending the FREContext class

An FREContext object provides a set of Java functions and context-specific state. Your extension must provide at least one concrete implementation of the FREContext class.

The FREContext class defines two abstract methods that you must implement:

- `getFunctions()` — must return a Map object, which the AIR runtime uses to look up the functions provided by your context.
- `dispose()` — called by the runtime when the context resources can be cleaned up.

Disposing a context

The ActionScript side of your extension can call the `dispose()` method of an ExtensionContext instance. Calling the ActionScript `dispose()` method causes the runtime to call the Java `dispose()` method of your FREContext class.

If the ActionScript side does not call `dispose()`, the runtime garbage collector disposes of the ExtensionContext instance when no more references to it exist. At that time, the runtime calls the Java `dispose()` method of your FREContext class.

FREContext example

The following example illustrates a simple FREContext implementation. The class creates a function map containing a single function.

```
package com.example;

import java.util.HashMap;
import java.util.Map;

import android.util.Log;

import com.adobe.fre.FREContext;
import com.adobe.fre.FREFunction;

public class ExtensionContext extends FREContext {
    private static final String CTX_NAME = "ExtensionContext";
    private String tag;

    public ExtensionContext( String extensionName ) {
        tag = extensionName + "." + CTX_NAME;
        Log.i(tag, "Creating context");
    }

    @Override
    public void dispose() {
        Log.i(tag, "Dispose context");
    }

    @Override
    public Map<String, FREFunction> getFunctions() {
        Log.i(tag, "Creating function Map");
        Map<String, FREFunction> functionMap = new HashMap<String, FREFunction>();

        functionMap.put( UsefulFunction.KEY, new UsefulFunction() );
        return functionMap;
    }

    public String getIdentifier() {
        return tag;
    }
}
```

Implementing the FREFunction interface

The AIR runtime uses the FREFunction interface to invoke your Java functions from ActionScript code. Implement this interface to expose a concrete piece of functionality.

The FREFunction interface defines a single method, `call()`. This method is called by the AIR runtime when ActionScript code in the extension invokes the `call()` method of the ExtensionContext class. To find the right FREFunction instance, the runtime looks in the FREContext function map. The ActionScript code passes an array of arguments to the method using the normal ActionScript types and classes for the arguments. Your Java code sees these values as FREObject instances. Likewise, your Java `call()` method returns an FREObject instance and the ActionScript code sees the return value as an ActionScript type.

When you implement an FREFunction class, you decide on the order of the parameters in the `call()` method arguments array. Since you also write the ActionScript side, you use that same parameter order in the ExtensionContext instance's `call()` method. Therefore, although every Java function parameter is an FREObject type, you know its corresponding ActionScript type.

Similarly you decide on the ActionScript type of the return value, if any, of a Java function. The `call()` method returns an object of this type. Although the Java function return value is always an FREObject instance, you know its corresponding ActionScript type.

The FREObject instances created by or passed to an FREFunction object have limited valid life spans. Do not save references to FREObjects between function invocations.

FREFunction example

The following example illustrates a simple FREFunction implementation. The function takes a boolean parameter and returns its negation.

```
package com.example;

import com.adobe.fre.FREContext;
import com.adobe.fre.FREFunction;
import com.adobe.fre.FREObject;

import android.util.Log;

public class UsefulFunction implements FREFunction {
    public static final String KEY = "usefulFunctionKey";
    private String tag;

    public FREObject call(FREContext arg0, FREObject[] arg1) {
        ExtensionContext ctx = (ExtensionContext) arg0;
        tag = ctx.getIdentifier() + "." + KEY;
        Log.i( tag, "Invoked " + KEY );
        FREObject returnValue = null;

        try {
            FREObject input = arg1[0];
            Boolean value = input.getAsBool();
            returnValue = FREObject.newObject( !value );//Invert the received value

        } catch (Exception e) {
            Log.d(tag, e.getMessage());
            e.printStackTrace();
        }
        return returnValue;
    }
}
```

Dispatching asynchronous events

The Java code can dispatch asynchronous events back to the ActionScript code of your extension. For example, an extension method can start another thread to perform some task. When the task in the other thread completes, that thread calls the `dispatchStatusEventAsync()` method of the appropriate FREContext. On the ActionScript side, the ActionScript ExtensionContext instance associated with that Java FREContext dispatches a Status event.

Accessing ActionScript objects

An instance of the FREObject class represents an ActionScript class object or primitive type. Use FREObject instances in your Java implementation to work with ActionScript data. A primary use of FREObjects is for Java function parameters and return values.

The FREObject class provides functions for accessing the associated ActionScript class object or primitive value. The Java APIs that you use depend on the type of the ActionScript object. These types can be the following:

- An ActionScript primitive data type
- An ActionScript class object
- An ActionScript String object
- An ActionScript Array or Vector class object
- An ActionScript ByteArray class object
- An ActionScript BitmapData class object

***Important:** You can only access an FREObject from the same thread as the one in which the "owning" FREFunction function is running. In addition, if you have called the `acquire()` method of any FREByteArray or FREBitmapData object, you cannot access the ActionScript methods of any object until you call the `release()` method for the original FREByteArray or FREBitmapData object. When any object is "acquired," calling the `getProperty()`, `setProperty()`, or `callMethod()` of that or any other FREObject throws an IllegalStateException.*

FREObject validity

If you attempt to use an invalid FREObject in a Java API call, the Java API throws an exception.

Any FREObject instance is valid only until the first FREFunction function on the call stack returns. The first FREFunction function on the call stack function is the one that the runtime calls due to the ActionScript side calling the ExtensionContext instance's `call()` method. FREObjects are also only valid in the thread used by the runtime to call the first FREFunction.

***Note:** An FREFunction function can indirectly call another FREFunction function. For example, `FREFunctionA()` can call a method of an ActionScript object. That method then can call `FREFunctionB()`.*

Therefore, when using an FREObject, consider the following:

- Any FREObject passed to an FREFunction instance is valid only until the first FREFunction on the call stack returns.
- Any FREObject that any Java function creates is valid only until the first FREFunction on the call stack returns.
- You cannot use an FREObject in another thread. Only use the FREObject in the same thread as the Java function that received or created the object.
- You cannot save an FREObject reference, for example in global data, between calls to FREFunction functions. Because the object becomes invalid when the first FREFunction function on the call stack returns, the saved reference is useless. However, you can use the `setActionScriptData()` method of the FREContext class to save data between function invocations.
- After an FREObject becomes invalid, the corresponding ActionScript object can still exist. For example, if an FREObject is a return value of an FREFunction function, its corresponding ActionScript object is still referenced. However, once the ActionScript side deletes its references, the runtime disposes of the ActionScript object.
- You cannot share FREObjects between extensions.

***Note:** You can share FREObjects between extension contexts of the same extension. However, in this, as in any case, the FREObject still becomes invalid when the first FREFunction function on the call stack returns to the runtime.*

Working with ActionScript primitive types and objects

Working with ActionScript primitive types

In your Java FREFunction implementations, an input parameter can correspond to a primitive ActionScript type. All parameters passed from ActionScript code to your extension are of type FREObject. Therefore, to work with an ActionScript primitive type input parameter, you get the ActionScript value of the FREObject parameter. Use the following FREObject functions:

- `getAsInt()`
- `getAsDouble()`
- `getAsString()`
- `getAsBool()`

If an output parameter or return value of an FREFunction corresponds to a primitive ActionScript type, you create the ActionScript primitive using a one of the FREObject factory methods. Use the following static FREObject functions:

- `FREObject.newObject( int value )`
- `FREObject.newObject( double value )`
- `FREObject.newObject( String value )`
- `FREObject.newObject( int boolean )`

Working with ActionScript Class objects

In your FREFunction implementations, an input parameter can correspond to an ActionScript class object. The FREObject class provides functions for accessing the ActionScript-defined properties and methods of the object:

- `getProperty( String propertyName )`
- `setProperty( String propertyName, FREObject value )`
- `callMethod( String propertyName, FREObject[] methodArgs )`

If an output parameter or return value corresponds to an ActionScript class object, you create the ActionScript object using a static FREObject factory method:

```
FREObject newObject ( String className, FREObject constructorArgs[] )
```

***Note:** These general ActionScript object manipulation functions apply to all ActionScript class objects. However, the ActionScript classes Array, Vector, ByteArray, and BitmapData are special cases because they each involve large amounts of data. Therefore, the Java API provides additional specific classes for manipulating objects of these special cases.*

Working with ActionScript ByteArray objects

Use the ActionScript ByteArray class to efficiently pass many bytes between the ActionScript side and Java side of your extension. The FREByteArray class extends FREObject with methods for handling byte array objects:

- `acquire()`
- `getLength()`

- `getBytes()`
- `release()`

To manipulate the bytes of the ByteArray object in the Java code, use the FREByteArray `acquire()` method, followed by the `getBytes()` method. After you have manipulated the bytes, use the FREByteArray `release()` method.

Do not call any FREObject methods (of any object, not just the acquired object) between the calls to `acquire()` and `release()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the byte array object or its contents.

Do not copy more bytes into the `bytes` field of the acquired FREByteArray object than the `length` field specifies. The ActionScript side will ignore any extra bytes.

You can create new FREByteArray object with the static factory method, `FREByteArray.newByteArray()`. The new byte array is empty. To add data to it from Java, you must first set the ActionScript `length` property:

```
FREByteArray freByteArray = freByteArray = FREByteArray.newByteArray();
freByteArray.setProperty( "length", FREObject.newObject( 12 ) );
freByteArray.acquire();
ByteBuffer bytes = freByteArray.getBytes();
//set the data
freByteArray.release();
```

Working with ActionScript Array and Vector objects

Use the ActionScript Vector and Array classes to efficiently pass arrays between the ActionScript side and Java side of your extension.

The FREArray class extends FREObject with the following methods appropriate for handling array and vector objects:

- `getLength()`
- `setLength( long length )`
- `getObjectAt( long index )`
- `setObjectAt( long index, FREObject value )`

The FREArray also defines factory methods for creating arrays and vectors:

- `FREArray newArray( int numElements )`
- `FREArray newArray( String classname, int numElements, boolean fixed )`

Working with ActionScript BitmapData objects

Use the ActionScript BitmapData class to pass bitmaps between the ActionScript side and Java side of your extension. The FREBitmapData class extends FREObject with the following methods appropriate for handling bitmap data objects:

- `acquire()`

  You must call `acquire()` before calling any other these other methods:
- `getHeight()`
- `getWidth()`
- `hasAlpha()`
- `isInvertedY()` (AIR 3.1)

- `isPremultiplied()`
- `getLineStride32()`
- `getBits()`
- `invalidateRect()`
- `release()`

To change the bitmap data, call `acquire()` and then get the pixel color data with the `getBits()` method. The `getLineStride32()` method indicates how many 32-bit data are in each horizontal line. Although this value is usually equal to the width of the bitmap, in some cases bitmap data can be padded with extra, non-visible bytes. Each pixel in the bitmap is a 32-bit value in ARGB format. After you have manipulated the bitmap, release the bitmap data by calling `release()`.

Do not call any FREObject methods (of any object, not just the acquired object) between the calls to `acquire()` and `release()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the byte array object or its contents.

To indicate that the Java implementation has modified all or part of the bitmap, invalidate a rectangle of the bitmap with the `invalidateRect()` function. The `x` and `y` parameters are the coordinates of the rectangle to invalidate, relative to the 0,0 coordinates which are the top, left corner of the bitmap. The `width` and `height` fields are the dimensions in pixels of the rectangle to invalidate.

You can create new FREBitmapData object with the static factory method, `FREBitmapData.newBitmapData()`.

Threads and native extensions

When coding your Java implementation, consider the following:

- The runtime can concurrently call an extension context's FREFunction functions on different threads.
- The runtime can concurrently call different extension contexts' FREFunction functions on different threads.

Therefore, code your Java implementations appropriately. For example, if you use global data, protect it with some form of locks.

***Note:** Your Java implementation can choose to create separate threads. If it does, consider the restrictions specified in "FREObject validity" on page 35.*

Chapter 5: Packaging a native extension

To provide your native extension to application developers, package all the related files into an ANE file. An AIR application developer uses the ANE file by:

- Including the ANE file in the application's library path in the same way the developer includes a SWC file in the library path. This action allows the application to reference the extension's ActionScript classes.
- Packaging the ANE file with the AIR application.

For information on building and packaging an ANE file with an AIR application, see Using native extensions for Adobe AIR.

Creating a native extension package involves the following tasks:

1. Build the extension's ActionScript library into a SWC file.
2. Build the extension's native libraries -- one for each supported target platform.
3. Create a signed certificate for your extension. Signing an extension is optional.
4. Create an extension descriptor file.
5. Use ADT to create the extension package.

Building the ActionScript library of a native extension

Build the ActionScript side of your extension into a SWC file. The SWC file is an ActionScript library — an archive file that contains your ActionScript classes and other resources, such as its images and strings.

When you package a native extension, you need both the SWC file and a separate library.swf file, which you extract from the SWC file. The SWC file provides the ActionScript definitions for authoring and compilation. The library.swf provides the ActionScript implementation used by a specific platform. If different target platforms of your extension require different ActionScript implementations, create multiple SWC libraries and extract the library.swf file separately for each platform. A best practice, however, is that all the ActionScript implementations have the same public interfaces. (Only one SWC file can be included in the ANE package.)

The SWC file contains a file called library.swf. For more information, see "The SWC file and SWF files in the ANE package" on page 49.

Use one of the following ways to build the SWC file:

- Use Adobe Flash Builder to create a Flex library project.

  When you build the Flex library project, Flash Builder creates a SWC file. See Create Flex library projects.

  Be sure to select the option to include Adobe AIR libraries when you create your Flex library project.

  Ensure that the SWC is compiled to the correct version of the SWF format. Use SWF 11 for AIR 2.7, SWF 13 for AIR 3, SWF 14 for AIR 3.1, and so on. You can set the SWF file format version in the project's properties. Select ActionScript Compiler and enter this Additional Compiler Argument:

  ```
  -swf-version 17
  ```

  Note: *You can use `swfdump` in the Flex SDK bin directory to check the SWF file format version of any SWF file:`swfdump myFlexLibraryProjectSWF.swf`*

- Use the command-line tool acompc to build a Flex library project for AIR. This tool is the component compiler provided with the Flex SDK. If you are not using Flash Builder, use acompc directly. See Using compc, the component compiler.

  For example:

```
acompc -source-path $HOME/myExtension/actionScript/src
                                -include-classes sample.extension.MyExtensionClass
sample.extension.MyExtensionHelperClass
                                -swf-version=13
                                -output
$HOME/myExtension/output/sample.extension.myExtension.swc
```

  Note: *If your ActionScript library uses any external resources, package them into the ANE file using ADT. See "Creating the native extension package" on page 46.*

SWF version compatibility

The SWF version specified when compiling the ActionScript library is one of the factors, along with extension descriptor namespace, that determines whether the extension is compatible with an AIR application. The SWF version of the extension cannot exceed the SWF version of the main application SWF file:

Compatible AIR application version	ANE SWF version	Extension namespace
3.0+	10-13	ns.adobe.com/air/extension/2.5
3.1+	14	ns.adobe.com/air/extension/3.1
3.2+	15	ns.adobe.com/air/extension/3.2
3.3+	16	ns.adobe.com/air/extension/3.3
3.4+	17	ns.adobe.com/air/extension/3.4
3.5+	18	ns.adobe.com/air/extension/3.5
3.6+	19	ns.adobe.com/air/extension/3.6
3.7+	20	ns.adobe.com/air/extension/3.7

Note: *The platform options (platform.xml) file requires the `ns.adobe.com/air/extension/3.1` namespace or later. If you are using the `-platformoptions` flag to package your ANE, you must specify `ns.adobe.com/air/extension/3.1` or later and a SWC version greater than or equal to 14. Some platform options file features require later AIR namespace and SWF versions.*

Creating a signed certificate for a native extension

You can choose to digitally sign your native extension. Signing an extension is optional.

Digitally signing your native extension with a certificate issued by a recognized certification authority (CA) provides significant assurance to AIR application developers that:

- the native extension they are packaging with their application has not been accidentally or maliciously altered.
- you are the signer (publisher) of the native extension.

Note: *When an AIR application is packaged, the AIR application's certificate, not the native extension's certificate if it is provided, is what the application's users see.*

Obtain a certificate from a certification authority. Creating a digitally signed certificate for your native extension is the same as creating a certificate for an AIR application. See Signing AIR applications in Building Adobe® AIR® Applications.

Creating the extension descriptor file

Each native extension contains an extension descriptor file. This XML file specifies information about the extension, such as the extension identifier, name, version number, and the platforms that it can run on.

When creating extensions, write the extension descriptor file according to the detailed schema "Native extension descriptor files" on page 72.

For example:

```
<extension xmlns="http://ns.adobe.com/air/extension/3.5">
                    <id>com.example.MyExtension</id>
                    <versionNumber>0.0.1</versionNumber>
                    <platforms>
                    <platform name="Android-ARM">
                    <applicationDeployment>
                    <nativeLibrary>MyExtension.jar</nativeLibrary>
                    <initializer>com.sample.ext.MyExtension</initializer>
                    </applicationDeployment>
                    </platform>
                    <platform name="iPhone-ARM">
                    <applicationDeployment>
                    <nativeLibrary>MyExtension.a</nativeLibrary>
                    <initializer>MyExtensionIntializer</initializer>
                    </applicationDeployment>
                    </platform>
                    <platform name="default">
                    <applicationDeployment/>
                    </platform>
                    </platforms>
                    </extension>
```

Consider the following information when creating your extension descriptor file.

The extension ID

The value of the `<id>` element is the same value used in:

- the ActionScript call to `CreateExtensionContext()`.
- the `extensionID` element in the application descriptor file of an application that uses the extension.

For best practices for naming the extension ID, see "The extension ID" on page 9.

The version number

The value of the `<versionNumber>` element is for specifying the version of the extension. One important use of the version number is for maintaining backward compatibility on device-bundled extensions. See "Native extension backward compatibility" on page 13.

Platforms

You can write a native extension that targets multiple platforms as discussed in "Targeting multiple platforms" on page 5.

Depending on the platform, the extension is either application-bundled or device-bundled, as discussed in "Extension availability at runtime" on page 5.

For each targeted platform, provide a `<platform>` element in the extension descriptor file. The `<platform>` element's `name` attribute specifies the target platform, such as `iPhone-ARM` or `Windows-x86`. Application-bundled extensions can also specify `default` for the `name` attribute value. This value indicates that the extension is an ActionScript-only extension; the extension has no native code libraries.

When an AIR application that uses an application-bundled extension runs, AIR does the following:

- AIR loads the extension libraries that the extension descriptor file associates with the platform name that corresponds to the device's platform.
- If no platform name corresponds to the device, AIR loads the extension libraries that the extension descriptor file associates with the default platform.

Descriptor namespace

The namespace specified in the root `<extension>` element of the descriptor file determines the version of the AIR SDK required by the extension. The namespace is one of the factors, along with SWF version, that determine whether an extension can be used in an AIR application. The AIR application descriptor namespace must be greater than or equal to the extension descriptor namespace.

Extension namespace value	Compatible AIR version	ANE SWF version
ns.adobe.com/air/extension/2.5	AIR 3+	13
ns.adobe.com/air/extension/3.1	AIR 3.1+	14
ns.adobe.com/air/extension/3.2	AIR 3.2+	15
ns.adobe.com/air/extension/3.3	AIR 3.3+	16
ns.adobe.com/air/extension/3.4	AIR 3.4+	17
ns.adobe.com/air/extension/3.5	AIR 3.5+	18
ns.adobe.com/air/extension/3.6	AIR 3.6+	19
ns.adobe.com/air/extension/3.7	AIR 3.7+	20

***Note:** The platform options (platform.xml) file requires the `ns.adobe.com/air/extension/3.1` namespace or later. If you are using the `-platformoptions` flag to package your ANE, you must specify `ns.adobe.com/air/extension/3.1` or later and a SWC version greater than or equal to 14. Some platform options file features require later AIR namespace and SWF versions.*

Building the native library

To build your extension's native library, use a development environment appropriate for the target device. For example:

- When developing with the Android SDK, use the Android Development tools plug-in for the Eclipse Integrated Development Environment (IDE).
- When developing for iOS devices and Mac OS X devices, use Apple's Xcode IDE.
- When developing for Windows devices, you can use Microsoft Visual Studio.

For native extension examples using these development environments, see Native extensions for Adobe AIR.

Build the native side of your extension into a library, not an application. When you package your native extension into an ANE file, you specify the native library.

Android native libraries

When you use the Android SDK, provide the library as a JAR file.

When you use the Android NDK, provide a shared library with a filename as follows:

```
lib<yourlibraryname>.so
```

This shared library naming convention is required for the application APK package to install correctly.

When you create an ANE package containing shared libraries, the libraries must be stored in the following directory structure:

```
<Android platform directory>/
                        libs/
                        armeabi/
                        <Android emulator native libraries>
                        armeabi-v7a/
                        <Android device native libraries>
```

iOS native libraries

Provide a static library with the .a filename extension. You can use the Cocoa Touch static library template in the Xcode IDE to build a .a file. Use the target type *device* to create a native library that runs on the device; use the target type *simulator* to create a native library that runs onthe iOS Simulator (iOS Simulator support in AIR 3.3 and higher).

Each version of AIR bundles a version of the iOS SDK with it. You can link to any of the public frameworks available in that version of the iOS SDK in a native extension that targets the corresponding AIR Version. The following table lists AIR SDK versions and their bundled iOSs versions, as well as support for additional features:

AIR SDK	**Included iOS SDK**	**link to additional libraries**	**bundle third-party libraries**
3.0 - 3.2	4.2	no	no
3.3 - 3.4	5.1	yes	no
3.5	6.0	yes	yes

When targetting a specific AIR SDK version with your extension, you should not use any iOS frameworks introduced after the corresponding iOS version. Unless the AIR SDK you're targetting , you should not use any other shared libraries or third-party frameworks.

As an alternative to using the iOS SDK that's bundled with the AIR SDK, in AIR 3.3 and later you can link to an external iOS SDK. Use the ADT `-platformsdk` switch, specifying the path to the external iOS SDK.

AIR links to the following iOS framework libraries by default:

AudioToolbox	CoreLocation	OpenGLES
AVFoundation	CoreMedia	QuartzCore
CFNetwork	CoreVideo	Security
CoreFoundation	Foundation	SystemConfiguration
CoreGraphics	MobileCoreServices	UIKit
GameController	AssetsLibrary	

When linking to other frameworks and libraries, specify the linkage options in a platform options XML file.

***Note:** In AIR 3.4 and later, you can use the* `ADT -hideAneSymbols yes` *option to eliminate potential symbol conflicts. For more information, see Native extension options.*

iOS platform options (platform.xml) file

You can use a platform options (platform.xml) file to specify iOS-specific options for linking to additional frameworks and libraries or for bundling third-party frameworks or libraries in your native extension. The platform options file is added to the ANE when you package the extension by specifying the `-platformoptions` flag after the `-platform` flag for iOS. Later, when a developer creates an application IPA file that uses your extension, ADT uses the options in the platform.xml file to link to the additional libraries and include the bundled dependencies.

Note: *The platform options file can have any name. You are not required to name it "platform.xml."*

You can use an iOS platform options file with both the iPhone-ARM (device) and iPhone-x86 (iOS simulator) platforms.

Platform options files require AIR 3.1 or later.

When using the *packagedDependencies* feature for packaging the ANE for iOS, add

```
<option>-rpath @executable_path/Frameworks</option>
```

inside the `linkerOptions` tag, in *platformoptions.xml*.

iOS linker options

The iOS linker options provides a way for you to pass arbitrary options to the linker. The specified options are passed unchanged to the linker. You can use this to link to additional frameworks and libraries, such as additional iOS frameworks. To specify linker options, use the `<linkerOptions>` tag in the platform options xml file. Inside the `<linkerOptions>` tag, specify each linkage option wrapped in an `<options>` tag pair, as shown in the following example:

```
<linkerOptions>
                                   <option>-ios_version_min 5.0</option>
                                   <option>-framework Accelerate</option>
                                   <option>-liconv</option>
                                   </linkerOptions>
```

Any dependency linked in this way must be distributed to developers who use your native extension separate from the native extension itself. This is most useful for using an additional library included with iOS but not linked by AIR by default. You can also use this option to allow developers to link to a static library that you provide them separately.

The `<linkerOptions>` tag requires AIR 3.3 or later.

Packaged third-party dependencies

In some cases you want to use a static library (for example, a native third-party library) in your native extension without having access to the source code for the library, or without requiring developers to have access to the library separate from your extension. You can bundle the static library with your native extension by specifying it as a packaged dependency. Use the `<packagedDependencies>` tag in the platform options xml file. For each dependency that you want to include in the extension package, specify its name or relative path surrounded in a `<packagedDependency>` tag pair, as shown in the following example:

```
<packagedDependencies>
                                   <packagedDependency>foo.a</packagedDependency>
                                   <packagedDependency>abc/x.framework</packagedDependency>
                                   <packagedDependency>lib.o</packagedDependency>
                                   </packagedDependencies>
```

A packaged dependency should be a static library with one of the following extensions: `.a`, `.framework`, or `.o`. The library should support the ARMv7 architecture to be used on a device, and the i386 architecture for use with the iOS simulator.

When packaging the native extension, you must specify the names of the dependencies as parameters to the `-platformoptions` flag. List the dependencies after the filename of the platform.xml file and before any following `-package` flag, as shown in the following example:

```
adt -package <signing options> -target ane MyExtension.ane MyExt.xml -swc MyExtension.swc
                                -platform iPhone-ARM -platformoptions platformiOSARM.xml
                                foo.a abc/x.framework lib.o -C platform/ios .
                                -platform iPhone-x86 -platformoptions platformiOSx86.xml
                                -C platform/iosSimulator
                                -platform default -C platform/default library.swf
```

The `<packagedDependencies>` tag requires AIR 3.5 or later.

Using private embebbed frameworks

To create iOS ANEs with Xcode6 or later, using private frameworks:

1 Edit platform.xml to add

   ```
   <option>-rpath @executable_path/Frameworks</option>
   ```

   inside the the `linkerOptions` tag.

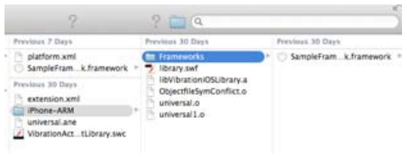
   For example,

   ```
   </description>
                                           <linkerOptions>
                                           <option>-ios_version_min 5.1.1</option>
                                           <option>-rpath
   @executable_path/Frameworks</option>
                                           </linkerOptions>
                                           <packagedDependencies>

   <packagedDependency>SampleFramework.framework</packagedDependency>
                                           </packagedDependencies>
   ```

2 Create a folder named, Frameworks, inside the existing iPhone-ARM folder .

3 Copy the private framework to the Frameworks folder and package the framework with the ANE.



Copy private framework

Platform options (platform.xml) file example

The following listing shows an example of the structure of a platform options (platform.xml) file:

```
<platform xmlns="http://ns.adobe.com/air/extension/3.5">
                                <description>An optional description.</description>
                                <copyright>2011 (optional)</copyright>
                                <sdkVersion>5.0.0</sdkVersion>
                                <linkerOptions>
                                <option>-ios_version_min 5.0</option>
                                <option>-framework Accelerate</option>
                                <option>-liconv</option>
                                </linkerOptions>
                                <packagedDependencies>
                                <packagedDependency>foo.a</packagedDependency>
                                <packagedDependency>abc/x.framework</packagedDependency>
                                <packagedDependency>lib.o</packagedDependency>
                                </packagedDependencies>
                                </platform>
```

To include this platform options file in the native extension package, you can use an ADT command such as the following:

```
adt -package <signing options> -target ane MyExtension.ane MyExt.xml -swc MyExtension.swc
                                -platform iPhone-ARM -platformoptions platformiOSARM.xml
                                foo.a abc/x.framework lib.o -C platform/ios .
                                -platform iPhone-x86 -platformoptions platformiOSx86.xml
                                -C platform/iosSimulator
                                -platform default -C platform/default library.swf
```

Mac OS X native libraries

For Mac OS X devices, provide a .framework library. When building the library, make sure that the Xcode project's setting for the base SDK is the Mac OS X 10.5 SDK.

When compiling Mac OS X frameworks for use as an extension, set the following options to support correct resolution of the dependency on the AIR framework in all scenarios:

- Add `@executable_path/../runtimes/air/mac`, `@executable_path/../Frameworks`, and `/Library/Frameworks` to LD_RUNPATH_SEARCH_PATHS.
- Use weak framework linking and the flat namespace option.

Together, these option settings allow the application to load the correct copy of the AIR framework first, and then for the extension to rely on the already loaded copy.

Windows native libraries

For Windows devices, provide the library as a DLL file. Dynamically link the library FlashExtensions.lib, located in the AIR SDK directory in the lib/windows directory, into your DLL. Also, if your native code library uses any of Microsoft's C runtime libraries, link to the multi-thread, static version of the C runtime library. To specify this type of linking, use the /MT compiler option.

Creating the native extension package

To provide your native extension to application developers, you package all the related files into an ANE file. The ANE file is an archive file that contains:

- The extension's ActionScript library

- The extension’s native code library
- The extension descriptor file
- The extension’s certificate
- The extension’s resources, such as images.

Use the AIR Developer Tool (ADT) to create the ANE file. Complete documentation for ADT is at AIR Developer Tool.

ADT example for packaging an extension

The following example illustrates how to package an ANE file with ADT. The example packages the ANE file to use with applications that:

- Run on Android devices.
- Run on Android x86 devices.
- Run on iOS devices.
- Run on the iOS Simulator.
- Run on other devices using the default ActionScript-only implementation.

```
adt -package <signing options> -target ane MyExtension.ane MyExt.xml -swc MyExtension.swc    -
platform Android-ARM -C platform/Android .
                        -platform Android-x86 -C platform/Android-x86 .
                        -platform iPhone-ARM -platformoptions platform.xml
                        abc/x.framework lib.o -C platform/ios .
                        -platform iPhone-x86 -C platform/iosSimulator
                        -platform default -C platform/default library.swf
```

In this example, the following command-line options are used to create the ANE package:

- *<signing options>*

  You can optionally sign the ANE file. For more information, see “Creating a signed certificate for a native extension” on page 40.

- `-target ane`

  The `-target` flag specifies which type of package to create. Use the `ane` target to package a native extension.

- `MyExtension.ane`

  Specify the name of the package file to create. Use the .ane filename extension.

- `MyExt.xml`

  Specify the extension descriptor file. The file specifies the extension ID and supported platforms. AIR uses this information to locate and load an extension for an application. In this example, the extension descriptor is:

```
<extension xmlns="http://ns.adobe.com/air/extension/3.1">
                                <id>com.sample.ext.MyExtension</id>
                                <versionNumber>0.0.1</versionNumber>
                                <platforms>
                                <platform name="Android-ARM">
                                <applicationDeployment>
                                <nativeLibrary>MyExtension.jar</nativeLibrary>
                                <initializer>com.sample.ext.MyExtension</initializer>
                                </applicationDeployment>
                                </platform>
                                <platform name="Android-x86">
                                <applicationDeployment>
                                <nativeLibrary>MyExtension.jar</nativeLibrary>
                                <initializer>com.sample.ext.MyExtension</initializer>
                                </applicationDeployment>
                                </platform>
                                <platform name="iPhone-ARM">
                                <applicationDeployment>
                                <nativeLibrary>MyExtension.a</nativeLibrary>
                                <initializer>InitMyExtension></initializer>
                                </applicationDeployment>
                                </platform>
                                <platform name="iPhone-x86">
                                <applicationDeployment>
                                <nativeLibrary>MyExtension.a</nativeLibrary>
                                <initializer>InitMyExtension></initializer>
                                </applicationDeployment>
                                </platform>
                                <platform name="default">
                                <applicationDeployment/>
                                </platform>
                                </platforms>
                                </extension>
```

See “Native extension descriptor files” on page 72 for more information.

- `MyExtension.swc`

  Specify the SWC file that contains the ActionScript side of the extension.

- `-platform Android-ARM -C platform/Android . -platform Android-x86 -C platform/Android-x86 .-platform iPhone-ARM -platformoptions platform.xml-Cplatform/iOS.`

  The `-platform` flag names a platform that this ANE file supports. The options that follow the name specify where to find the platform-specific libraries and resources. In this case, the `-C` option for the `Android-ARM` platform indicates to make the relative directory `platform/Android` the working directory. The directories and files that follow are relative to the new working directory.

  Therefore, in this example, the relative directory `platform/Android` contains all the Android native code library and resources. It also contains the Android platform-specific library.swf file and any other Android platform-specific SWF files.

  The `-platformoptions` flag for the iPhone-ARM platform is an optional entry that lets you specify platform-specific options. These options include options for linking to iOS frameworks (other than the default ones) as well as bundling third-party static libraries with your native extension. See “iOS native libraries” on page 43.

- `-platform default -C platform/default library.swf`

When the `-platform` option names the `default` platform, do not specify any native code files. Specify only a library.swf file, as well as other SWF files, if any.

The SWC file and SWF files in the ANE package

You specify a SWC file in the `-swc` option of the ADT packaging command. This SWC file is your ActionScript library. It contains a file called library.swf. ADT puts the library.swf from the SWC file into the ANE file. When an AIR application uses a native extension, it includes the extension's ANE file in its library path so that the application can compile. In fact, the application compiles against the public interfaces in library.swf.

Sometimes you create a different ActionScript implementation for each target platform. In this case, compile a SWC file for each platform and put the library.swf file from each SWC file in the appropriate platform directory before using the ADT packaging command. You can extract library.swf from a SWC file with extraction tools such as WinZip.

Consider the case when your extension's public interfaces are the same across all platforms. In this case, it doesn't matter which of the platform-specific SWC files you use in the `-swc` option of the ADT command. It doesn't matter because the SWF file in the SWC file is used only for application compilation. The SWF file that you put in the platform directory is the file that the native extension uses when it runs.

Note the following about the library.swf file:

- ADT requires that you provide a main SWF file named library.swf for each platform. When you create a SWC file, library.swf is the name of the SWF file.
- The library.swf file for each platform can be different.
- The library.swf file for each platform can be the same if the ActionScript side has no platform dependencies.
- The library.swf for each platform can load other SWF files that you include in the platform-specific directory. These other SWF files can have any name.

ANE file rules for application packaging

Use ADT to package applications that use native extensions. When you package an application that uses an extension, ADT verifies that there is a platform specified in the ANE file that matches the application packaging target . For example, the platform `Android-ARM` matches an Android apk package.

Furthermore, ADT matches the `default` platform with *any* target package. The `default` platform specifies an ActionScript-only version of the extension. Consider an AIR application that uses an application-bundled extension. AIR loads the ActionScript library of the `default` platform extension only if none of the extension's specified platforms correspond to the device.

For example, consider an application-bundled extension that specifies the platforms `iPhone-ARM`, `Android-ARM`, and `default`. When the application using the extension runs on a Windows platform, it uses the extension's `default` platform library.

Therefore, when you create an ANE file for application-bundling, consider the following rules that ADT uses when packaging an application that uses the ANE file:

- To create an Android application package, the ANE file must include the `Android-ARM` platform. Alternatively, the ANE file must include the default platform and at least one other platform.
- To create an iOS application package, the ANE file must include the `iPhone-ARM` platform. Alternatively, the ANE file must include the default platform and at least one other platform.
- To create an iOS Simulator application package, the ANE file must include the `iPhone-x86` platform. Alternatively, the ANE file must include the default platform and at least one other platform.

- To create a Mac OS X application package, the ANE file must include the `MacOS-x86-64` platform. Alternatively, the ANE file must include the default platform and at least one other platform.
- To create a Windows application package, the ANE file must include the `Windows-x86` platform. Alternatively, the ANE file must include the default platform and at least one other platform.

Only one platform implementation is ever bundled with an application. However, when you test an application containing an extension using the ADL utility, then the implementation is chosen at run time. This run-time selection can lead to differences in behavior depending on the test platform and the ANE package. For example, if the ANE includes implementations for the `Android-ARM`, `Windows-x86`, and `default` platforms, the implementation used when testing is different depending on whether the test computer is running Windows or OS X. On Windows, the `Windows-x86` platform implementation is used (even when testing under the mobile profile); on OS X, the `default` implementation is used.

Including additional Android shared .so libraries in the ANE package

Consider a native extension that targets the platform `Android-ARM`. The primary library of the native side of the extension is either:

- a .so library if you use the Android NDK
- a JAR file if you use the Android SDK

Sometimes, however, the native side requires more native libraries (.so libraries) than the primary .so library or JAR file for the extension.

For example:

- You are using the Java API to develop your native extension. However, you want to use the JNI (Java Native Interface) to access native .so libraries from your Java code.
- You are using the C API to develop your native extension. However, you want to partition your code into multiple shared libraries. Based on your extension's logic, you want to load the appropriate shared .so library at runtime.

When you create the ANE package, use the following directory structure:

```
<Android platform directory>/
                        libs/
                        armeabi/
                        <Android emulator native libraries>
                        armeabi-v7a/
                        <Android device native libraries>
```

When you use ADT to create the ANE package, set the `-platform` option to specify the Android platform directory contents:

```
-platform Android-ARM -C <Android platform directory> .
```

When an application developer uses ADT to include the ANE package in an APK package, the APK package includes the following libraries:

- The libraries in `libs/armeabi-v7a` if the ADT target is `apk` or `apk-captive-runtime`.
- The libraries in `libs/armeabi` if the ADT target is `apk-emulator`, `apk-debug`, or `apk-profile`.

***Note:** For the iPhone-ARM platform, you cannot include shared libraries in your ANE file. For more information, see "Building the native library" on page 42.*

Including resources in your native extension package

The ActionScript side and the native code side of your extension sometimes use external resources, such as images.

On the ActionScript side, your platform-independent SWC file can include the resources it requires. If a platform-dependent SWF file requires resources, include the resources in the platform-dependent directory structure that you specify in the ADT command.

On the native code side, you can also include resources in the platform-dependent directory structure. Put the resources in a subdirectory relative to the native code library, in the location the native code expects. ADT preserves this directory structure when packaging your ANE.

Using resources on Android and iOS devices have some extra requirements.

Resources on Android devices

For the `Android-ARM` platform, put the resources in a subdirectory called `res`, relative to the directory containing the native code library. Populate the directory with resources as described at Providing Resources on developer.android.com. When ADT packages the ANE file, it puts the resources in the `Android-ARM/res` directory of the resulting ANE package.

When accessing a resource from the extension's Java native code library, use the `getResourceID()` method in the FREContext class. Do not access the resources using the standard Android resource ID mechanism. For more information on the `getResourceID()` method, see "Method details" on page 129.

The method `getResourceId()` takes a String parameter that is the resource name. To avoid resource name conflicts between extensions in an application, provide a unique name for each resource file in an extension. For example, prefix the resource name with the extension ID, creating a resource name such as `com.sample.ext.MyExtension.myImage1.png`.

Accessing native resources with R.* mechanism

In previous releases, the native Android resources in the Android Native Extension could only be accessed by using the `getResourceID()` API while the R.* mechanism could not be used with the ANEs. Beginning AIR 4.0, you can access resources with the R.* mechanism. When using the R.* mechanism, ensure that you use the platform descriptor, platform.xml, which has all the dependencies defined:

```
<?xml version="1.0"?>
                            <xs:schema
                            xmlns:xs="http://www.w3.org/2001/XMLSchema"
                            targetNamespace="http://ns.adobe.com/air/extension/4.0"
                            xmlns="http://ns.adobe.com/air/extension/4.0"
                            elementFormDefault="qualified">
                            <xs:element name="platform">
                            <xs:complexType>
                            <xs:all>
                            <xs:element name="description"   type="LocalizableType"
minOccurs="0"/>
                            <xs:element name="copyright"     type="xs:string"
minOccurs="0"/>
                            <xs:element name="packagedDependencies" minOccurs="0">
                            <xs:complexType>
                            <xs:all>
                            <xs:element name="packagedDependency" type="name" minOccurs="0"
maxOccurs="unbounded"/>
                            </xs:all>
                            </xs:complexType>
                            </xs:element>
                            <xs:element name="packagedResources" minOccurs="0">
                            <xs:complexType>
                            <xs:all>
                            <xs:element name="packagedResource" minOccurs="0"
maxOccurs="unbounded"/>
                            <xs:complexType>
                            <xs:all>
                            <xs:element name="packageName" type="name" minOccurs="0"/>
                            <xs:element name="folderName" type="name" minOccurs="0"/>
                            </xs:all>
                            </xs:complexType>
                            </xs:all>
                            </xs:element>
                            </xs:complexType>
                            </xs:element>
                            </xs:all>
                            </xs:complexType>
                            </xs:element>
                            <xs:simpleType name="name">
                            <xs:restriction base="xs:string">
                            <xs:pattern value="[A-Za-z0-9\-\.]{1,255}"/>
                            </xs:restriction>
                            </xs:simpleType>
                            <xs:complexType name="LocalizableType" mixed="true">
                            <xs:sequence>
                            <xs:element name="text" minOccurs="0" maxOccurs="unbounded">
                            <xs:complexType>
                            <xs:simpleContent>
                            <xs:extension base="xs:string">
                            <xs:attribute name="lang" type="xs:language" use="required"/>
                            </xs:extension>
                            </xs:simpleContent>
                            </xs:complexType>
                            </xs:element>
                            </xs:sequence>
                            </xs:complexType>
                            </xs:schema>
```

Following is an example of the dependencies in `platform.xml`:

```
<packagedDependencies>
                            <packagedDependency>android-support-v4.jar</packagedDependency>
                            <packagedDependency>google-play-
services.jar</packagedDependency>
                            </packagedDependencies>
                            <packagedResources>
                            <packagedResource>
                            <packageName>com.myane.sampleasextension</packageName>
                            <folderName>ane-res</folderName>
                            </packagedResource>
                            <packagedResource>
                            <packageName>com.google.android.gms</packageName>
                            <folderName>google-play-services-res</folderName>
                            </packagedResource>
                            </packagedResources>
```

where:

- `packagedDependencies` is used to provide the name of all the jars on which the ANE is dependent.
- `packagedResources` defines the resources used by the ANE or any other jar file.
- `folderName` defines the name of the resources folder.
- `packageName` defines the package name of the jar that uses the resources.
- `packagedDependencies` and `packagedResources` will be available from extension namespace 4.0 onwards.

The Android-ARM folder contains all the ANE jar files and resources as well as the third-party jar files. Following is a sample ANE packaging command:

```
bin/adt -package -target ane sample.ane extension.xml -swc sampleane.swc -platform Android-ARM
-platformoptions platform.xml -C Android-ARM .
```

You do not need to merge the third-party jar files and resources with the ANE jar and resources when using the R.* resource access mechanism. ADT merges the jars and resources internally. All dependencies and resources still need to be packaged in the ANE.

Note that:

- An ANE project should be a library project for the R.* resource access mechanism to work.
- There is no restriction on the name of the resource folder, it can either start with 'res' or any other valid string.
- If the `-platformoptions` switch is not used while packaging an ANE, resource access is only possible via the `getResourceId()` mechanism.

Resources on iOS devices

Resource location

Before using ADT to create an ANE file for iOS devices, put the non-localized resources in the directory containing the native code library. Localized resources go in subdirectories as described in the next section.

However, an iOS application bundle contains its resources at the top level of the application bundle directory. These resources include all the resources used by the platform-specific part of each extension. When an AIR application developer packages an ANE file with the application, ADT does the following:

1. It looks in the ANE package's `iPhone-ARM` directory.
2. It considers all files in that directory, except the library.swf and the extension descriptor file, as resource files.
3. It moves the resource files to the top-level directory of the application.

Because resource files from multiple extensions are moved to the same location, resource filename conflicts are possible. If a conflict exists, ADT does not package the application and reports the errors. Therefore, provide a unique name for each resource file in an extension. For example, prefix the name with the extension ID, creating a resource name such as `com.sample.ext.MyExtension.myImage1.png`.

***Note:** The resources are in the top-level application directory -- not in the extension directory. Therefore, to access the resources, use the ActionScript property `File.applicationDirectory`. Do not use the ActionScript API `ExtensionContext.getExtensionDirectory()` to navigate to the extension directory to find the resources. They are not there.*

Localized resources

Your extension's native code library (but not the ActionScript side) can use localized resources. To use localized resources, do the following:

- In the platform-specific directory that you specify when creating your ANE file using ADT, put the localized resources in language-specific subdirectories. Name each of these subdirectories as follows:

  `language``[_`*`region`*`].lproj`

  Set the value of *`language`* and *`region`* according to iOS conventions. See Language and Locale Designations in the iOS Developer Library.

- Put localized strings into .strings files in the format described at developer.apple.com/library/ios/navigation. However, do not name any file `Localizable.strings` because that is the default name used in an application bundle.

The following directory structure is an example of a directory that contains all the iOS platform-specific files to package into the ANE file:

```
platform/
                            iOS/
                            library.swf
                            myNativeLibrary.a
                            myNonLocalizedImage1.jpg
                            de.lproj/
                            MyGermanLocalizable.strings
                            MyGermanLocalizedImage1.png
                            en_us.lproj/
                            MyAmericanLocalizable.strings
                            MyAmericanLocalizedImage1.png
                            en_gb.lproj/
                            MyBritishLocalizable.strings
                            MyBritishLocalizedImage1.png
```

Chapter 6: Building and installing native extensions for AIR for TV

To develop native extensions for an AIR for TV device, you must have access to the AIR for TV extensions development kit (EDK). The EDK is distributed to device manufacturers and system-on-a-chip manufacturers who include AIR for TV with their product.

For more information for manufacturers using AIR for TV, see *Getting Started with Adobe AIR for TV (PDF)*

Overview of tasks in developing AIR for TV extensions

When you develop a native extension for an AIR for TV device, do the following iterative tasks:

1 Write the native implementation.

For more information, see "Coding the native side with C" on page 15.

2 Write the real ActionScript implementation.

For more information, see "Coding the ActionScript side" on page 7. Be sure to consider "Native extension backward compatibility" on page 13.

3 Write the stub ActionScript implementation, and optionally write a simulator ActionScript implementation.

For more information, see "The device-bundled extension and the stub extension" on page 62 and "Check for extension support" on page 63.

4 Create a certificate for signing your native extension. This step is optional.

For more information, see "Creating a signed certificate for a native extension" on page 40.

5 Build the device-bundled extension and the stub extension by using the AIR for TV make utility. This process creates a ZIP file to install the device-bundled extension on the device. It also creates an ANE file of the stub extension for an AIR application to build and test with.

For more information, see "Building an AIR for TV native extension" on page 64.

6 If a simulator ActionScript implementation is available, build the simulator extension by using the AIR for TV make utility. This process creates an ANE file for an AIR application to build and test with.

For more information, see "Building an AIR for TV native extension" on page 64.

7 Add any required resources, such as images, to the ZIP file and ANE file.

For more information, see "Adding resources to your AIR for TV native extension" on page 70.

8 Test the simulator extension on a desktop computer.

For more information, see Debugging AIR for TV applications.

9 Install the device-bundled extension on the device.

For more information, see "Installing the device-bundled extension on the AIR for TV device" on page 70.

10 Test the device-bundled extension on the device.

For more information, see "Running an AIR application on an AIR for TV device" on page 71 and Debugging AIR for TV applications.

11 Deliver the stub or simulator ANE files, or both, to application developers.

AIR for TV extension examples

Adobe® AIR® for TV provides several examples of native extensions. The native implementation is written in C++ and uses the AIR for TV extensions development kit (EDK). The EDK is distributed to device manufacturers and system-on-a-chip manufacturers who include AIR for TV with their product. More information about the AIR for TV EDK and creating AIR for TV extensions is in "Building an AIR for TV native extension" on page 64.

As an AIR for TV extension developer, you can:

- Copy these examples as starting point for your extension.
- See these examples for code samples that show how to use various C API extension functions as well as the ActionScript ExtensionContext class.
- Copy the makefile of one of these examples as a starting point for creating the makefile for your extension.

HelloWorld example

The HelloWorld example is in the following directory of the AIR for TV distribution:

```
<AIR for TV installation directory>/products/stagecraft/source/ae/edk/helloworld
```

The HelloWorld example is a simple extension to illustrate basic extension behavior. It does the following:

- Uses the `ExtensionContext.call()` method to pass a string from the ActionScript side to the native implementation.
- Returns a string from the native implementation to the ActionScript side.
- Starts a thread in the native implementation that sends asynchronous events to the ActionScript side.

The following table describes each file and its location relative to the `helloworld/` directory:

File	**Description**
HelloWorld.as in directory as/device/src/tv/adobe/extension/example/	ActionScript side of the real (not the stub) extension that defines the HelloWorld class. It does the following: • Creates an ExtensionContext instance. • Defines the extension's ActionScript APIs: `Hello()` and `StartCount()`. • Listens for events on the ExtensionContext instance, and redispatches the events to the HelloWorld instance's listeners.
HelloWorld.as in directory as/distributable/src/tv/adobe/extension/example/	ActionScript of the stub extension that defines the HelloWorld class. This ActionScript-only stub defines the extension's ActionScript APIs: `Hello()` and `StartCount()`. However, the stub implementations do not call native functions.
HelloWorldExtensionClient.as in directory client/src	AIR application that uses the extension. The AIR application is the client of the extension. It does the following: • Creates an instance of the HelloWorld class. • Listens for events on the HelloWorld instance. • Calls the HelloWorld instance's `Hello()` and `StartCount()` APIs.
HelloWorldExtensionClient-app.xml in directory client/src	AIR application descriptor file. Includes the `<extensions>` element with the `extensionID` value `tv.adobe.extension.example.HelloWorld`.
HelloWorld.h in directory native/	The C++ header file of the HelloWorld class.
HelloWorld.cpp in directory native/	The C++ implementation file of the HelloWorld class. The implementation does the following: • Defines the FREFunction `Hello()` which writes its string parameter to the console. It also returns the string "Hello from extensionland". • Defines the FREFunction `StartCount()` which starts an asynchronous thread to send one event every 500 milliseconds to the ExtensionContext instance.
HelloWorldExtension.cpp in directory native/	Contains implementations of the following C API extension functions: • `FREInitializer()` • `FREContextInitializer()` • `FREContextFinalizer()` • `FREFinalizer()`
PlatformEDKExtension_HelloWorld.mk	The makefile for the HelloWorld extension.

File	Description
ExtensionUtil.h in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	Contains macros convenient to coding your C or C++ implementation.
ExtensionBridge.cpp in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	The AIR for TV extension module implementation. When you build your native implementation, include this source file in your build.
phonyEdkAneCert.p12 in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	A phony certificate that the make utility uses for packaging the HelloWorld extension into an ANE file.
extension.mk in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	The makefile for the extension module. Do not modify this file.

Process example

The Process example is in the following directory of the AIR for TV distribution:

```
<AIR for TV installation directory>/products/stagecraft/source/ae/edk/process
```

The Process extension allows an AIR application to manipulate Linux processes, including the following functionality:

- Spawning a Linux process. The process executes a Linux command that the AIR application specifies.
- Getting the process identifier of the process.
- Checking whether the process has completed.
- Receiving an event that the process has completed.
- Getting the return code from the completed process.
- Receiving events indicating that the process has written to `stdout` or `stderr`.
- Retrieving the output strings from `stdout` and `stderr`.
- Writing strings to `stdin`.
- Sending an interrupt signal to the process.
- Killing the process.

The following table describes each file and its location relative to the `process/` directory:

File	**Description**
Process.as in directory as/device/src/tv/adobe/extension/process/example/	ActionScript side of the real (not the stub) extension that defines the Process class. It does the following: • Creates an ExtensionContext instance. • Defines the extension's ActionScript APIs. • Listens for events on the ExtensionContext instance, and redispatches the events to the Process instance's listeners.
ProcessEvent.as in directory as/device/src/tv/adobe/extension/process/example/	Defines the ProcessEvent class, which derives from the Event class. The AIR application ActionScript listens for these ProcessEvent notifications.
Process.as in directory as/distributable/src/tv/adobe/extension/process/example/	ActionScript of the stub extension that defines the Process class. This ActionScript-only stub defines the extension's ActionScript APIs. However, the stub implementations do not call native functions.
ProcessExtensionClient.as in directory client/simple/src	AIR application that uses the extension. The AIR application is the client of the extension. It provides an example of how an AIR application uses the Process extension APIs.
ProcessExtensionClient-app.xml in directory client/simple/src	AIR application descriptor file. Includes the `<extensions>` element with the `extensionID` value `tv.adobe.extension.process.Process`.
Process.h in directory native/	The C++ header file of the abstract Process class.
ProcessLinux.h in directory native/	The C++ header file of the concrete ProcessLinux class. The ProcessLinux class derives from the Process class. It declares private methods and data for a Linux implementation of the Process class.
ProcessLinux.cpp in directory native/	The C++ implementation file of the ProcessLinux class. The implementation includes the following functionality: • Defines the FREFunction functions. These functions use Linux system calls to, for example, fork and exec a Linux process and to interact with `stdin`, `stdout`, and `stderr` • Monitors the status of the spawned process. The implementation creates a thread for this purpose. The thread uses the C extension API `FREDispatchStatusEventAsync()` to report events. • Defines helper functions for creating FREObject variables for returning information from the FREFunction functions to the ActionScript side. These helper functions use C API extension functions such as `FRENewObjectFromBool()`, `FRENewObjectFromUTF8()`, and `FRENewObjectFromUint32()`.

File	**Description**
ProcessExtension.cpp in directory native/	Contains implementations of the following C API extension functions: • `FREInitializer()` • `FREContextInitializer()` • `FREContextFinalizer()` • `FREFinalizer()`
PlatformEDKExtension_Process.mk	The makefile for the Process extension.
ExtensionUtil.h in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	Contains macros convenient to coding your C or C++ implementation.
ExtensionBridge.cpp in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	The AIR for TV extension module implementation. When you build your native implementation, include this source file in your build.
phonyEdkAneCert in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	A phony certificate that the make utility uses for packaging the Process extension into an ANE file.
extension.mk in directory *<AIR for TV installation directory>*/products/stagecraft/source/ae/edk	The makefile for the extension module. Do not modify this file.

Build the extension examples on the x86Desktop platform

To build the Hello World or Process extension example on the x86Desktop platform, first build your AIR for TV distribution for the x86Desktop platform. Instructions are in *Getting Started with Adobe AIR for TV (PDF)* in *Quick Start on Linux*.

Note: *Make sure that you installed the AIR 3 SDK, the The Open Source Flex® SDK, and the Java™ runtime, and included the bin directories in your PATH environment variable. Building native extensions for AIR for TV depends on these libraries.*

Next, add building the Hello World or Process extension example to your x86Desktop build. Do the following:

1 Change to the directory:

```
<AIR for TV installation directory>/products/stagecraft/build/linux/platforms/x86Desktop
```

2 Link to the .mk file of the extension you want to build. For example:

```
ln -s <AIR for TV installation
directory>/products/stagecraft/source/ae/edk/helloworld/PlatformEDKExtension_HelloWorld.mk
```

3 Change to the directory:

```
<AIR for TV installation directory>/products/stagecraft/build/linux
```

4 Build the distribution, including the extension example:

```
make
```

Alternatively, build only the extension example. For example:

```
make PlatformEDKExtension_HelloWorld.mk
```

The make utility creates two files for each extension example. It puts the files in one of the following directories depending on whether you specified debug or release for `SC_BUILD_MODE`:

```
<AIR for TV installation directory>/build/stagecraft/linux/x86Desktop/debug/bin
<AIR for TV installation directory>/build/stagecraft/linux/x86Desktop/release/bin
```

The files that the make utility creates for the example extension are:

- A ZIP file that contains the device-bundled extension.
- An ANE file that contains the stub or simulator extension.

The device-bundled extension and the stub extension

When you write a native extension for an Adobe® AIR® for TV device, you are required to create two variations of the extension:

- The device-bundled extension, also called the real extension.
- The stub extension.

Furthermore, you can optionally provide a third variation: the simulator extension.

The device-bundled extension

The device-bundled extension is the variation that is installed on the device. The ActionScript side calls functions of the native implementation. You build this real ActionScript implementation, along with the native implementation, creating a ZIP file. The device manufacturer unzips this file into a specific directory on the device.

The stub extension

A stub native extension has the same ActionScript interfaces as the real ActionScript implementation, but the ActionScript methods don't do anything. The stub extension is an ActionScript-only extension; it has no native implementation. When you build the stub ActionScript implementation, you create an ANE file.

AIR application developers use this ANE file for three purposes:

- To compile an AIR application that uses the native extension.
- To run the AIR application on a desktop computer, rather than on the target device.
- To include in the AIR application package.

The simulator extension

An optional third variation is a simulator extension. This implementation also has the same ActionScript interfaces as the real ActionScript implementation. However, its ActionScript methods simulate the extension's behavior in ActionScript. Like the stub extension, the simulator extension is an ActionScript-only extension; it has no native implementation. When you build the simulator ActionScript implementation, you create an ANE file.

AIR application developers can compile their applications using the simulator extension ANE file. They can use this ANE file to test the application on a desktop computer more thoroughly than testing with the stub extension. They can also include the simulator extension in the AIR application package.

***Note:** You can create a simulator extension in place of, or in addition to, the stub extension.*

Use of the device-bundled, stub, and simulator extensions

An AIR application developer does the following with the stub and simulator extensions:

- Uses the stub extension or simulator extension to compile the AIR application.
- Uses the stub extension or simulator extension to test the application on a desktop computer.
- Packages the stub extension or simulator extension into their distributable AIR application.

  ***Note:** If you provide both a stub and simulator extension to the AIR application developers, instruct them about which one to package with their distributable application.*

When the AIR application runs on the device, AIR for TV does the following:

1. Looks for the corresponding device-bundled (real) extension on the device.
2. If it is there, AIR for TV loads it for the AIR application to use.
3. If it is not there, AIR for TV instead loads the stub or simulator extension packaged with the application.

Check for extension support

A best practice is to provide a handshake between the native extension and the AIR for TV application. The handshake tells the application about the availability of the extension on the device. This information allows the AIR for TV application to decide what logic paths to take.

Decide what public interfaces to provide in your ActionScript extension class for this handshake. Then, instruct AIR for TV application developers on the use of these interfaces.

For example, consider a native extension that has:

- A stub extension for packaging with the AIR for TV application.
- A simulator extension for testing the application on a desktop computer.
- The device-bundled, real extension for installing on the target device.

In the ActionScript side of each variation, define a method called `isSupported()` with the following implementations:

- In the real ActionScript implementation that interacts with the native implementation, implement the method to return `true`.

  The calling AIR application is running on a device on which the extension is device-bundled. Therefore, the application knows it can continue using the extension.

- In the stub ActionScript implementation, implement the method to return `false`.

  The calling AIR application is running on a device on which the extension is not device-bundled. In this situation, AIR for TV uses the stub extension that was packaged with the AIR application. The `false` return value informs the application not to call any other methods of the extension. The application can make the appropriate logic decisions, such as to exit gracefully.

- In a simulator ActionScript implementation, implement the method to return `true`.

The calling AIR application is running on a desktop computer for testing purposes. Therefore, the `true` return value informs the application that it can continue using the extension.

However, depending on your extension, you can instruct application developers to package their applications with your simulator extension. Then, if the application is running on a device that does not have the device-bundled extension, the application proceeds with the simulator version of the extension.

Building an AIR for TV native extension

When you build an AIR for TV native extension, you build two versions of the extension:

- The device-bundled extension.
- The stub or simulator extension.

The device-bundled extension includes:

- A native implementation, typically written in C or C++.
- The real ActionScript implementation that calls functions of the native implementation.

The stub or simulator extension is an ActionScript-only implementation.

For more information on the real, stub, and simulator ActionScript implementations, see "The device-bundled extension and the stub extension" on page 62.

Creating a signed certificate for the extension

You can choose to digitally sign your native extension. Signing an extension is optional.

The AIR for TV make utility uses a phony certificate by default. This phony certificate is useful only for testing. For information about creating a certificate authority certificate, see "Creating a signed certificate for a native extension" on page 40.

Writing the native implementation

For AIR for TV, the native implementation of your extension is an AIR for TV module. See *Optimizing and Integrating Adobe AIR for TV (PDF)* for general information about AIR for TV modules, including details about building the modules.

The AIR for TV distribution provides an extension development kit (EDK) for writing and building the native implementation of your extension.

The EDK includes the following:

- The C extension API header file:

  ```
  <AIR for TV installation
  directory>/products/stagecraft/include/ae/edk/FlashRuntimeExtensions.h
  ```

  This header file declares the C types and functions that the native implementation uses.

- An extension module implementation in the following source file:

  ```
  <AIR for TV installation directory>/products/stagecraft/source/ae/edk/ExtensionBridge.cpp
  ```

  Do not modify this extension module implementation. When you build your native implementation, you must include this source file in your build.

- Makefile support to build your device-bundled extension. For more information, see "Creating the .mk file" on page 65 and "Running the make utility" on page 68.

Note: *The AIR for TV EDK requires that the* `FREInitializer()` *method is named* `Initializer()` *and that the* `FREFinalizer()` *method is named* `Finalizer()`*. For more information about these methods, see "Extension initialization" on page 15 and "Extension finalization" on page 19.*

Placing ActionScript and native code in the directory structure

Device-bundled extensions are specific to a hardware platform. When you develop a device-bundled extension, place your files in a subdirectory for your platform. This subdirectory is in the following directory:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/<yourCompany>/stagecraft-platforms/<yourPlatform>/edk
```

For example, CompanyA uses the following subdirectory for development targeting their PlatformB:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/CompanyA/stagecraft-platforms/PlatformB/edk
```

Put the header and source files for your C implementation in the `<yourPlatform>/edk` directory or its subdirectories. For example, put your extension .cpp and .h files in the following directory:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/native
```

Similarly, put the .as files for your real ActionScript implementation in the `<yourPlatform>/edk` directory or its subdirectories. For example:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/as/real
```

Also, put the .as files for your stub or simulator ActionScript implementation in the `<yourPlatform>/edk` directory or its subdirectories. For example:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/as/stub
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/as/simulator
```

Note: *The sample extensions that AIR for TV provides are in the directory* `<AIR for TV installation directory>/products/stagecraft/source/edk`*. Do not put your extension files in this directory.*

Creating the .mk file

As with other AIR for TV modules, to build your extensions module, you first create a .mk file. The primary purpose of the .mk file is to specify the source files to build.

To create the .mk file:

1 Copy the PlatformEDKExtension_HelloWorld.mk file or PlatformEDKExtension_Process.mk file from:

```
<AIR for TV installation directory>/products/stagecraft/source/ae/edk/helloworld/
```

or

```
<AIR for TV installation directory>/products/stagecraft/source/ae/edk/process/
```

Copy it to:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/<yourCompany>/stagecraft-platforms/<yourPlatform>
```

This directory is the same directory that contains your platform's Makefile.config file. See *Creating your platform Makefile.config* in the chapter *Coding, building, and testing* in *Optimizing and Integrating Adobe AIR for TV (PDF)*.

2 Rename the .mk file to PlatformEDKExtension_<*your extension name*>.mk. The AIR for TV make utility automatically discovers .mk files with this naming convention.

   Always start the .mk file's name with PlatformEDKExtension_.

3 Edit the sections of the .mk file that are marked "REQUIRED".

Make the following required modifications:

- Set `SC_EDK_EXTENSION_NAME` to the extension name. Set this variable to the value of *`<your extension name>`* in PlatformEDKExtension_<*your extension name*>.mk.
- Set `SC_EDK_EXTENSION_PACKAGE` to the extension package name. Set this value to the package name you use in the ActionScript side of your extension.

  The make utility uses this value as the value of the `<id>` element in the extension's extension descriptor file. It also names the resulting ANE file with this value and the .ane filename extension.

  For more information about the extension descriptor file, see "Native extension descriptor files" on page 72.
- Set `SC_EDK_EXTENSION_VERSION` to the version number of the extension.

  The make utility uses this value as the value of the `<versionNumber>` element in the extension's extension descriptor file.
- Set `SC_MODULE_SOURCE_DIR`, `SC_MODULE_SOURCE_FILES`, and `SC_ADDITIONAL_MODULE_OBJ_SUBDIRS` to specify the native implementation files that AIR for TV provides.

  Note: *Do not remove ExtensionBridge.cpp from this list. Remove the HelloWorld or Process extension implementation files. Typically, you do not add your extension's source files to this list.*

  For example:

  ```
  SC_MODULE_SOURCE_DIR  := $(SC_SOURCE_DIR_EDK)
  SC_MODULE_SOURCE_FILES  := ExtensionBridge.cpp
  ```
- Set `SC_PLATFORM_SOURCE_DIR` and `SC_PLATFORM_SOURCE_FILES` to specify the native implementation files of your extension. For example:

  ```
  SC_PLATFORM_SOURCE_DIR:= $(SC_PLATFORM_MAKEFILE_DIR)/edk/myExtension/native
  SC_PLATFORM_SOURCE_FILES := \
                  MyExtension.cpp \
                  helper\MyHelperClass1.cpp \
                  helper\MyHelperClass2.cpp
  ```
- Set `SC_EDK_AS_SOURCE_DIR` to the directory that contains the ActionScript files for the real (not the stub) implementation of your extension. For example:

  ```
  SC_EDK_AS_SOURCE_DIR  := $(SC_PLATFORM_MAKEFILE_DIR)/edk/myExtension/as/real
  ```

  Note: *This directory is the base directory for your ActionScript package. For example, consider an ActionScript package named `tv.adobe.extension.example`. The directories `tv`, `adobe`, `extension`, and `example` are successive subdirectories of `SC_EDK_AS_SOURCE_DIR`.*
- Set `SC_EDK_AS_CLASSES` to list every ActionScript class that the real ActionScript implementation defines. For example:

```
SC_EDK_AS_CLASSES := MyExtension \
                    MyHelperClass1 \
                    MyHelperClass2
```

- Set `SC_EDK_AS_SOURCE_DIR_AUTHORING` to the directory that contains the ActionScript files for the stub or simulator implementation of your extension. For example:

  ```
  SC_EDK_AS_SOURCE_DIR_AUTHORING := $(SC_PLATFORM_MAKEFILE_DIR)/edk/myExtension/as/stub
  ```

  ***Note:** This directory is the base directory for your ActionScript package. For example, consider an ActionScript package named `tv.adobe.extension.example`. The directories `tv`, `adobe`, `extension`, and `example` are successive subdirectories of `SC_EDK_AS_SOURCE_DIR_AUTHORING`.*

- Set `SC_EDK_AS_CLASSES_AUTHORING` to list every ActionScript class that the stub or simulator ActionScript implementation defines. For example:

  ```
  SC_EDK_AS_CLASSES_AUTHORING := MyExtension \
                    MyHelperClass1 \
                    MyHelperClass2
  ```

Install third-party libraries

Building AIR for TV requires some third-party libraries. Details on these libraries are in *Install third-party software* in *Getting Started with Adobe AIR for TV (PDF)*.

If you are building only your extension module, and not building all of AIR for TV, the necessary libraries are:

- The AIR 3 SDK

  Select the Mac OS X download from http://www.adobe.com/products/air/sdk/.

  Create a directory to contain the .tbz2 file's contents. For example:

  ```
  /usr/AIRSDK
  ```

  Extract the .tbz2 file's contents into the directory.

  ```
  tar jxf AdobeAIRSDK.tbz2
  ```

  Set the `PATH` environment variable to include the AIR SDK bin directory. In this example, the bin directory is /usr/AIRSDK/bin.

- The Open Source Flex® SDK.

  Download the ZIP file of the latest release build of the Open Source Flex SDK from http://opensource.adobe.com/wiki/display/flexsdk/Downloads.

  Create a directory to contain the ZIP file's contents. For example:

  ```
  /usr/flexSDK
  ```

  Extract the ZIP file's contents into the directory.

  ```
  unzip flex_sdk_4.5.1.21328_mpl.zip
  ```

  Set the `PATH` environment variable to include the Flex SDK bin directory. In this example, the bin directory is /usr/flexSDK/bin.

- The Java™ runtime. The Flex SDK requires a recent Java runtime. If your development system does not yet have a Java runtime, downloads and installation instructions are at http://www.java.com/en/download/manual.jsp.

  Set the `PATH` environment variable to include the Java bin directory.

Running the make utility

Details on using the AIR for TV make utility are in:

- *Getting Started with Adobe AIR for TV (PDF)* in the chapter *Installing and building the source distribution*.
- *Optimizing and Integrating Adobe AIR for TV (PDF)*in the chapter *Coding, building, and testing*. Follow the instructions in the section called *Creating your platform Makefile.config* to set the various build variables.

In particular, the make utility uses the following build variables in Makefile.config when building extensions:

- `SC_ZIP`
- `SC_UNZIP`
- `SC_PLATFORM_NAME`
- `SC_PLATFORM_ARCH`

After you have created your platform's Makefile.config file and your extension's .mk file, you can use the make utility to do the following:

- Build all of AIR for TV.
- Build only your extension module.

To build all of AIR for TV, do the following:

1. Make sure the environment variables `SC_BUILD_MODE` and `SC_PLATFORM` are set.
2. If you are using a certificate that you created to sign your extension, set the environment variables `SC_EDK_ANE_CERT_FILE` and `SC_EDK_ANE_CERT_PASSWD`.

   Set `SC_EDK_ANE_CERT_FILE` to the relative or absolute path to your certificate. The relative path is relative to the build directory <*AIR for TV installation directory*>/stagecraft/build/linux.

   Set `SC_EDK_ANE_CERT_PASSWD` to the password of the certificate.

   If you do not set these environment variables, the make utility uses a default phony certificate, and displays a warning message. This phony certificate is appropriate only for your testing.
3. Change to the directory:

   ```
   <AIR for TV installation directory>/products/stagecraft/build/linux
   ```
4. Enter the following command:

   ```
   make
   ```

To build only your extension module, do the following:

1. Make sure the environment variables `SC_BUILD_MODE` and `SC_PLATFORM` are set.
2. If you are using a certificate that you created to sign your extension, set the environment variables `SC_EDK_ANE_CERT_FILE` and `SC_EDK_ANE_CERT_PASSWD` as described in the previous steps.
3. Change to the directory stagecraft/build/linux.
4. Enter the following command:

   ```
   make PlatformEDKExtension_<your extension name>
   ```

You can remove all objects previously built for your extension with the following command:

```
make clean-PlatformEDKExtension_<your extension name>
```

You can remove all objects previously built for your extension and then rebuild them with the following command:

```
make rebuild-PlatformEDKExtension_<your extension name>
```

Important: *The make utility sometimes fails if your build machine is behind a firewall. A firewall can prohibit access to the time-stamp server that ADT uses when it packages the native extension into an ANE file. This failure results in the following error output:*

```
Could not generate timestamp: Connection timed out
```

To work around this failure, modify the ADT command that the make utility uses. Edit the file extension.mk in the following directory:

```
<AIR for TV installation directory>/stagecraft/source/ae/edk/
```

Find the following line:

```
$(SC_EXEC_CMD) $(SC_ADT) -package \
```

Add the parameter `-tsa none` to the command as follows:

```
$(SC_EXEC_CMD) $(SC_ADT) -package -tsa none\
```

Make utility extension output

The make utility creates two files for your extension. It puts the files in one of the following directories depending on whether you specified debug or release for `SC_BUILD_MODE`:

```
<AIR for TV installation directory>/build/stagecraft/linux/<yourPlatform>/debug/bin
<AIR for TV installation directory>/build/stagecraft/linux/<yourPlatform>/release/bin
```

The files that the make utility creates for your extension are:

- A ZIP file that contains the device-bundled extension to deploy on the device.
- An ANE file that contains the stub or simulator extension. AIR application developers use this ANE file to build their applications. They also use it to test their applications on a desktop computer using ADL. They also package this ANE file with their application into an AIRN package.

Building both a stub and simulator extension

Sometimes you want to build both a stub and a simulator extension, in addition to the real extension. Typically, you instruct the AIR application developers to do the following:

- Test on the desktop using the simulator extension.
- Package the stub extension with their application into an AIRN package.

To build both a stub and simulator extension, do the following:

1 Create the stub extension and its .mk file. Make sure you can build the stub extension and the real extension.

2 Create a directory for your simulator implementation that is a sibling of your stub implementation directory. For example:

```
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/as/stub
<AIR for TV installation directory>/products/stagecraft/thirdparty-
private/CompanyA/stagecraft-platforms/PlatformB/edk/myExtension/as/simulator
```

3 Make a copy of the .mk file for your extension.

4 In the copy, edit the values of `SC_EDK_AS_SOURCE_DIR_AUTHORING` and `SC_EDK_AS_CLASSES_AUTHORING`. Set the values to reflect your simulator implementation directory and classes.

5 Rename the original .mk file for your extension to keep it safe. Then, rename the copy to the .mk filename for your extension: PlatformEDKExtension_<*your extension name*>.mk.

6 Move the stub ANE file in your platform's bin directory to a safe place. Otherwise, the next step overwrites it.

7 Run the make utility to build the real extension and your simulator extension.

Adding resources to your AIR for TV native extension

Some AIR for TV native extensions require resources, such as image files. If your extension uses resources, do the following:

1 Extract the files from the ZIP file that the make utility created, preserving the directory structure.

2 Add resource directories and files that the extension requires to the directory structure.

3 Rezip the updated directory structure into a ZIP file with the same name as the ZIP file that the make utility created.

4 Extract the files from the ANE file that the make utility created, preserving the directory structure. You can use the same tool you use to handle ZIP files.

5 Add resource directories and files that the extension requires to the directory structure.

6 Rezip the updated directory structure into a ZIP file. Rename the file to the same name as the ANE file that the make utility created. Make sure that you change the filename extension to .ane.

The exact subdirectory structure for your resources within the extension's directory structure depends on where your extension looks for the resources. The ActionScript side of the extension can use `ExtensionContext.getExtensionDirectory()` to locate the extension's directory. For more information, see "Access the native extension's directory" on page 12.

Distributing the AIR for TV native extension

Installing the device-bundled extension on the AIR for TV device

After you have built the device-bundled extension, install it on the device.

The make utility created a ZIP file that contains all the files to install on the device. Unzip this file onto the device into the following location:

```
/opt/adobe/stagecraft/extensions
```

***Note:** The /opt/adobe/stagecraft/extensions directory can also be a Linux symbolic soft link to another directory on the device filesystem. For more information, see the chapter `Filesystem usage` in Optimizing and Integrating Adobe AIR for TV (PDF).*

Packaging the stub or simulator extension with an AIR application

For an AIR application to use a device-bundled extension on a device, package the stub or simulator extension with the AIR application. Use ADT to package the stub or simulator extension's ANE file with the AIR application to create an AIRN package file. An AIRN package file is just like a AIR package file, but an AIRN package file includes an extension ANE file.

For more information, see *Packaging an AIR for TV application*.

Running an AIR application on an AIR for TV device

See *Getting Started with Adobe AIR for TV (PDF)* for how to install and run an AIR application on an AIR for TV device. Make sure that you use the following command-line option when you run the stagecraft binary executable:

```
--profile extendedTV
```

This option is required for the AIR application to be able to use its native extensions.

When AIR for TV runs the application that contains a stub or simulator extension, AIR for TV looks for corresponding device-bundled extension on the device. If it is there, the AIR application uses it. If the device-bundled extension is not installed on the device, the AIR application uses the stub or simulator extension.

Chapter 7: Native extension descriptor files

An extension descriptor file describes the contents of a native extension package.

Example extension descriptor

The following extension descriptor document describes a native extension for:

- an Android device
- a *default* ActionScript implementation for other platforms

```
<extension xmlns="http://ns.adobe.com/air/extension/3.1">
    <id>com.example.MyExtension</id>
    <versionNumber>0.0.1</versionNumber>
    <platforms>
        <platform name="Android-ARM">
            <applicationDeployment>
                <nativeLibrary>MyExtension.jar</nativeLibrary>
                <initializer>com.sample.ext.MyExtension</initializer>
            </applicationDeployment>
        <platform name="default">
            <applicationDeployment/>
        </platform>
    </platforms>
</extension>
```

The extension descriptor file structure

The extension descriptor file is an XML document with the following structure:

```
<extension xmlns="http://ns.adobe.com/air/extension/2.5">
    <id>...</id>
    <versionNumber>...</versionNumber>
    <name>
        <text xml:lang="language_code">...</text>
    </name>
    <description>
        <text xml:lang="language_code">...</text>
    </description>
    <platforms>
        <platform name="device">
            <applicationDeployment>
                <nativeLibrary>...</nativeLibrary>
                <initializer>...</initializer>
                <finalizer>...</finalizer>
            </applicationDeployment>
        </platform>
        <platform name="device">
            <deviceDeployment/>
        <platform name="default">
            <applicationDeployment/>
    </platforms>
</extension>
```

Native extension descriptor elements

The following dictionary of elements describes each of the legal elements of an AIR application descriptor file.

applicationDeployment

Declares a native code library included in the extension package and, hence, deployed with the application.

Each `platform` element must contain either the `applicationDeployment` element or the `deviceDeployment` element, but not both.

Parent elements:platform.

Child elements:

- finalizer
- initializer
- nativeLibrary

Content

Identifies the native code library and the initialization and finalization functions. When the platform name is `default`, the `applicationDeployment` element has no child elements because the `default` platform has no native code libraries.

Example

```
<applicationDeployment>
    <nativeLibrary>myExtension.so</nativeLibrary>
    <initializer>com.example.extension.Initializer</initializer>
    <finalizer>com.example.extension.Finalizer</finalizer>
</applicationDeployment>
```

copyright

Optional

A copyright declaration for the extension.

Parent elements:extension

Child elements: none

Content

A string containing copyright information.

Example

```
<copyright>© 2010, Examples, Inc. All rights reserved.</copyright>
```

description

Optional

The description of the extension.

Parent elements:extension

Child elements:text

Content

Use a simple text node or multiple `text` elements.

Using multiple `text` elements, you can specify multiple languages in the `description` element. The `xml:lang` attribute for each text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

Example

Description with simple text node:

```
<description>This is a sample native extension for Adobe AIR.</description>
```

Description with localized text elements for English, French, and Spanish:

```
<description>
    <text xml:lang="en">This is a example.</text>
    <text xml:lang="fr">C'est un exemple.</text>
    <text xml:lang="es">Esto es un ejemplo.</text>
</description>
```

deviceDeployment

Declares a native extension for which the code libraries are deployed separately on the device and are not included in this extension package.

Device deployment is not supported by all platforms.

Each `platform` element must contain either the `applicationDeployment` element or the `deviceDeployment` element, but not both.

Parent elements:platform

Child elements: None.

Content

None. The `deviceDeployment` element must be empty.

Example

```
<deviceDeployment/>
```

extension

Required

The root element of the extension descriptor document.

Parent elements: None.

Child elements:

- copyright
- description
- id
- name
- platforms
- versionNumber

Content

Identifies the supported platforms and the code libraries for each platform.

The `extension` element contains a namespace attribute called `xmlns`. Set the `xmlns` value to one of the following values:

```
xmlns="http://ns.adobe.com/air/extension/3.1"
xmlns="http://ns.adobe.com/air/extension/2.5"
```

The namespace is one of the factors, along with the SWF version, that determines compatibility between an AIR SDK and an ANE file. The AIR application must be packaged with a version of the AIR SDK that equals or exceeds the extension namespace. Thus an AIR 3 application can use an extension with the 2.5 namespace, but not the 3.1 namespace.

Example

```
<extension xmlns="http://ns.adobe.com/air/extension/2.5">
    <id>com.example.MyExtension</id>
    <versionNumber>1.0.1</versionNumber>
    <platforms>
        <platform name="Polyphonic-MIPS">
            <deviceDeployment/>
        </platform>
        <platform name="NeoTech-ARM">
            <deviceDeployment/>
        </platform>
        <platform name="Philsung-x86">
            <deviceDeployment/>
        </platform>
        <platform name="default">
            <applicationDeployment/>
        </platform>
    </platforms>
</extension>
```

finalizer

Optional

The finalization function defined in the native library.

Parent elements:applicationDeployment

Child elements: None.

Content

The name of the finalizer function if the extension uses the C API in its native library.

If the extension uses the Java API, this element contains the name of the class that implements the FREExtension interface.

The value can contain these characters: A - Z, a - z, 0 - 9, period (.), and dash (-).

Example

```
<finalizer>...</finalizer>
```

id

Required

The ID of the extension.

Parent elements:extension

Child elements: None.

Content

Specifies the ID of the extension.

The value can contain these characters: A - Z, a - z, 0 - 9, period (.), and dash (-).

Example

```
<id>com.example.MyExtension</id>
```

initializer

Optional

The initialization function defined in the native library. An initializer element is required if the `nativeLibrary` element is used.

Parent elements:applicationDeployment

Child elements: None.

Content

The name of the initialization function if the extension uses the C API in its native library.

If the extension uses the Java API, this element contains the name of the class that implements the FREExtension interface.

The value can contain these characters: A - Z, a - z, 0 - 9, period (.), and dash (-).

Example

```
<initializer>...</initializer>
```

name

Optional

The name of the extension.

Parent elements:extension

Child elements:text

Content

If you specify a single text node (instead of multiple `<text>` elements), the AIR application installer uses this name, regardless of the system language.

The `xml:lang` attribute for each text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

Example

The following example defines a name with a simple text node:

```
<name>Test Extension</name>
```

The following example, specifies the name in three languages (English, French, and Spanish) using <text> element nodes:

```
<name>
    <text xml:lang="en">Hello AIR</text>
    <text xml:lang="fr">Bonjour AIR</text>
    <text xml:lang="es">Hola AIR</text>
</name>
```

nativeLibrary

Optional

The native library file included in the extension package for a platform. Consider the following:

- The `nativeLibrary` element is not required if the extension contains only ActionScript code.
- If the `nativeLibrary` element is not used, the `initializer` and `finalizer` elements cannot be used either.
- If the `nativeLibrary` element is used, the `initializer` element is also required.

Parent elements:applicationDeployment

Child elements: None.

Content

The filename of the native library included in the extension package.

The value can contain these characters: A - Z, a - z, 0 - 9, period (.), and dash (-).

Example

```
<nativeLibrary>extensioncode.so</nativeLibrary>
```

platform

Required

Specifies the native code library for the extension on a specific platform.

Parent elements:platforms

Child elements: One, and only one, of the following elements:

- applicationDeployment
- deviceDeployment

Content

The `name` attribute specifies the name of the platform. The special `default` platform name allows the extension developer to include an ActionScript library that simulates the behavior of the native code on unsupported platforms. Simulated behavior can be used to support debugging and to provide fall back behavior for multi-platform applications.

Use the following values for the `name` attribute:

- `Android-ARM` for Android devices.
- `default`
- `iPhone-ARM` for iOS devices.
- `iPhone-x86` for the iOS Simulator.
- `MacOS-x86-64` for Mac OS X devices.
- `QNX-ARM` for Blackberry Tablet OS devices.
- `Windows-x86` for Windows devices.

***Note:** Device-bundled extensions use a `name` attribute value defined by the device manufacturer.*

The child elements specify how the native code library is deployed. Application deployment means that the code library is deployed with each AIR application that uses it. The code library must be included in the extension package. Device deployment means that the code library is deployed separately to the platform and is not included in the extension package. The two deployment types are mutually exclusive; include only one deployment element.

Example

```
<platform name="Philsung-x86">
    <deviceDeployment/>
</platform>
<platform name="default">
    <applicationDeployment/>
</platform>
```

platforms

Required

Specifies the platforms supported by this extension.

Parent elements:extension

Child elements:platform

Content

A `platform` element for each supported platform. Optionally, a special *default* platform can be specified containing an ActionScript implementation for use on platforms not supported with a specific code library.

Example

```
<platforms>
    <platform name="Android-ARM">
        <applicationDeployment>
            <nativeLibrary>MyExtension.jar</nativeLibrary>
            <initializer>com.sample.ext.MyExtension</initializer>
            <finalizer>com.sample.ext.MyExtension</finalizer>
        </applicationDeployment>
    </platform>
    <platform name="iPhone-ARM">
        <applicationDeployment>
            <nativeLibrary>MyExtension.a</nativeLibrary>
            <initializer>InitMyExtension></initializer>
        </applicationDeployment>
    <platform name="Philsung-x86">
        <deviceDeployment/>
    </platform>
    <platform name="default">
        <applicationDeployment/>
    </platform>
</platforms>
```

text

Optional

Specifies a localized string.

The `xml:lang` attribute of a text element specifies a language code, as defined in RFC4646 (http://www.ietf.org/rfc/rfc4646.txt).

AIR uses the `text` element with the `xml:lang` attribute value that most closely matches the user interface language of the user's operating system.

For example, consider an installation in which a `text` element includes a value for the en (English) locale. AIR uses the en name if the operating system identifies en (English) as the user interface language. It also uses the en name if the system user interface language is en-US (U.S. English). However, if the user interface language is en-US and the application descriptor file defines both en-US and en-GB names, then the AIR application installer uses the en-US value.

If the application defines no `text` element that matches the system user interface languages, AIR uses the first `name` value defined in the extension descriptor file.

Parent elements:

- name
- description

Child elements: none

Content

An `xml:lang` attribute specifying a locale and a string of localized text.

Example

```
<text xml:lang="fr">Bonjour AIR</text>
```

versionNumber

Required

The extension version number.

Parent elements:extension

Child elements: none

Content

The version number can contain a sequence of up to three integers separated by periods. Each integer must be a number from 0 to 999 (inclusive).

Examples

```
<versionNumber>1.0.657</versionNumber>

<versionNumber>10</versionNumber>

<versionNumber>0.01</versionNumber>
```

Chapter 8: Native C API Reference

For native extension examples using the native C API, see Native extensions for Adobe AIR.

Typedefs

FREContext

AIR 3.0 and later

```
typedef void* FREContext;
```

The ActionScript side creates an extension context by calling `ExtensionContext.createExtensionContext()`. For each extension context, the AIR runtime creates a corresponding FREContext variable.

The runtime passes the FREContext variable to each of the following functions in the native implementation:

- The context initialization function, FREContextInitializer().
- The context finalizer function, FREContextFinalizer().
- Each extension function, FREFunction().

Use this FREContext variable when:

- You dispatch an event to the ActionScript ExtensionContext instance. See FREDispatchStatusEventAsync().
- You get or set native context data. See "FREGetContextNativeData()" on page 99 and "FRESetContextNativeData()" on page 114.
- You get or set ActionScript context data. See "FREGetContextActionScriptData()" on page 98 and "FRESetContextActionScriptData()" on page 113.

FREObject

AIR 3.0 and later

```
typedef void* FREObject;
```

When you access an ActionScript class object or primitive data type variable from your native C implementation, you use an FREObject variable. The runtime associates an FREObject variable with the corresponding ActionScript object.

FREObject variables are used in native functions you implement with the signature FREFunction(). The native functions:

- Receive FREObject variables as parameters.
- Return an FREObject variable.

FREObject variables are also used when you use native extensions C API functions to:

- Create an ActionScript class object or ActionScript primitive data type.
- Get the value of an ActionScript class object or ActionScript primitive data type.
- Create an ActionScript String object.

- Get the value of an ActionScript String object.
- Get or set a property of an ActionScript object.
- Call a method of an ActionScript object.
- Access the bits of an ActionScript BitmapData object.
- Access the bytes of an ActionScript ByteArray object.
- Get or set the length of an ActionScript Array or Vector object.
- Get or set an element of an ActionScript Array or Vector object.
- Get an ActionScript Error object when the runtime throws an exception in a native extension C API function call.
- Set or get an ActionScript object in the ActionScript context data.

Structure typedefs

FREBitmapData

AIR 3.0

Use the FREBitmapData or FREBitmapData2 structures when acquiring and manipulating the bits in an ActionScript BitmapData class object. The FREBitmapData structure is defined as follows:

```
typedef struct {
    uint32_t    width;
    uint32_t    height;
    uint32_t    hasAlpha;
    uint32_t    isPremultiplied;
    uint32_t    lineStride32;
    uint32_t*   bits32;
} FREBitmapData;
```

The fields of FREBitmapData have the following meanings:

width A uint32_t that specifies the width, in pixels, of the bitmap. This value corresponds to the `width` property of the ActionScript BitmapData class object. This field is read-only.

height A uint32_t that specifies the height, in pixels, of the bitmap. This value corresponds to the `height` property of the ActionScript BitmapData class object. This field is read-only.

hasAlpha A uint32_t that indicates whether the bitmap supports per-pixel transparency. This value corresponds to the `transparent` property of the ActionScript BitmapData class object. If the value is non-zero, then the pixel format is `ARGB32`. If the value is zero, the pixel format is `_RGB32`. Whether the value is big endian or little endian depends on the host device. This field is read-only.

isPremultiplied A uint32_t that indicates whether the bitmap pixels are stored as premultiplied color values. A non-zero value means the values *are* premultipled. This field is read-only. For more information about premultiplied color values, see BitmapData.getPixel() in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

lineStride32 A uint32_t that specifies the number of uint32_t values per scanline. This value is typically the same as the `width` parameter. This field is read-only.

bits32 A pointer to a uint32_t. This value is an array of uint32_t values. Each value is one pixel of the bitmap.

Note: *The only field of a* `FREBitmapData` *structure that you can change in the native implementation is the* `bits32` *field. The* `bits32` *field contains the actual bitmap values. Treat all the other fields in the* `FREBitmapData` *structure as read-only fields.*

More Help topics

"FREBitmapData2" on page 83

"FREAcquireBitmapData()" on page 91

"FREInvalidateBitmapDataRect()" on page 105

FREBitmapData2

AIR 3.1 and later

The FREBitmapData2 structure adds the `isInvertedY` field to the FREBitmapData structure. In other respects, the two structures are identical. The FREBitmapData2 structure is defined as follows:

```
typedef struct {
    uint32_t    width;
    uint32_t    height;
    uint32_t    hasAlpha;
    uint32_t    isPremultiplied;
    uint32_t    lineStride32;
    uint32_t    isInvertedY
    uint32_t*   bits32;
} FREBitmapData2;
```

The fields of FREBitmapData2 have the following meanings:

width A uint32_t that specifies the width, in pixels, of the bitmap. This value corresponds to the `width` property of the ActionScript BitmapData class object. This field is read-only.

height A uint32_t that specifies the height, in pixels, of the bitmap. This value corresponds to the `height` property of the ActionScript BitmapData class object. This field is read-only.

hasAlpha A uint32_t that indicates whether the bitmap supports per-pixel transparency. This value corresponds to the `transparent` property of the ActionScript BitmapData class object. If the value is non-zero, then the pixel format is `ARGB32`. If the value is zero, the pixel format is `_RGB32`. Whether the value is big endian or little endian depends on the host device. This field is read-only.

isPremultiplied A uint32_t that indicates whether the bitmap pixels are stored as premultiplied color values. A non-zero value means the values *are* premultipled. This field is read-only. For more information about premultiplied color values, see BitmapData.getPixel() in the *ActionScript 3.0 Reference for the Adobe Flash Platform*.

lineStride32 A uint32_t that specifies the number of uint32_t values per scanline. This value is typically the same as the `width` parameter. This field is read-only.

isInvertedY A uint32_t that indicates the order in which the rows of bitmap data in the image are stored. A non-zero value means that the bottom row of the image appears first in the image data (in other words, the first value in the `bits32` array is the first pixel of the last row in the image). A zero value means that the top row of the image appears first in the image data. This field is read-only.

bits32 A pointer to a uint32_t. This value is an array of uint32_t values. Each value is one pixel of the bitmap.

***Note:** The only field of a `FREBitmapData2` structure that you can change in the native implementation is the `bits32` field. The `bits32` field contains the actual bitmap values. Treat all the other fields in the `FREBitmapData2` structure as read-only fields.*

More Help topics

"FREBitmapData" on page 82

"FREAcquireBitmapData2()" on page 92

"FREInvalidateBitmapDataRect()" on page 105

FREByteArray

AIR 3.0 and later

Use the FREByteArray structure when acquiring and manipulating the bytes in an ActionScript ByteArray class object. The structure is defined as follows:

```
typedef struct {
    uint32_t    length;
    uint8_t*    bytes;
} FREByteArray;
```

The fields of FREByteArray have the following meanings:

length A uint32_t that is the number of bytes in the `bytes` array.

bytes A uint8_t* that is a pointer to the bytes in the ActionScript ByteArray object.

More Help topics

"FREAcquireByteArray()" on page 93

FRENamedFunction

AIR 3.0 and later

Use the FRENamedFunction to associate an FREFunction that you write with a name. Use that name in your ActionScript code when calling the native function with the ExtensionContext instance `call()` method. The structure is defined as follows:

```
typedef struct FRENamedFunction_{
    const uint8_t*  name;
    void*           functionData;
    FREFunction     function;
} FRENamedFunction;
```

The fields of FRENamedFunction have the following meanings:

name A const uint8_t*. This pointer points to a string that the ActionScript side uses to call the associated C function. That is, the string value is the name that the ActionScript ExtensionContext `call()` method uses in its `functionName` parameter. Use UTF-8 encoding for the string and terminate it with the null character.

functionData A void*. This pointer points to any data you want to associate with this FREFunction function. When the runtime calls the FREFunction function, it passes the function this data pointer.

function An FRENamedFunction. The function that the runtime associates with the string given by the `name` field. Define this function with the signature of an FREFunction().

Enumerations

FREObjectType

AIR 3.0 and later

An FREObject variable corresponds to an ActionScript class object or primitive type. The FREObjectType enumeration defines values for these ActionScript class types and primitive types. The C API function FREGetObjectType() returns an FREObjectType enumeration value that best describes an FREObject variable's corresponding ActionScript class object or primitive type.

```
enum FREObjectType {
    FRE_TYPE_OBJECT             = 0,
    FRE_TYPE_NUMBER             = 1,
    FRE_TYPE_STRING             = 2,
    FRE_TYPE_BYTEARRAY          = 3,
    FRE_TYPE_ARRAY              = 4,
    FRE_TYPE_VECTOR             = 5,
    FRE_TYPE_BITMAPDATA         = 6,
    FRE_TYPE_BOOLEAN            = 7,
    FRE_TYPE_NULL               = 8,
    FREObjectType_ENUMPADDING   = 0xfffff
};
```

The enumeration values have the following meanings:

FRE_TYPE_OBJECT The FREObject variable corresponds to an ActionScript class object that is not a String object, ByteArray object, Array object, Vector object, or BitmapData object.

FRE_TYPE_NUMBER The FREObject variable corresponds to an ActionScript Number variable.

FRE_TYPE_STRING The FREObject variable corresponds to an ActionScript String object.

FRE_TYPE_BYTEARRAY The FREObject variable corresponds to an ActionScript ByteArray object.

FRE_TYPE_ARRAY The FREObject variable corresponds to an ActionScript Array object.

FRE_TYPE_VECTOR The FREObject variable corresponds to an ActionScript Vector object.

FRE_TYPE_BITMAPDATA The FREObject variable corresponds to an ActionScript BitmapData object.

FRE_TYPE_BOOLEAN The FREObject variable corresponds to an ActionScript Boolean variable.

FRE_TYPE_NULL The FREObject variable corresponds to the ActionScript value `Null` or `undefined`.

FREObjectType_ENUMPADDING This final enumeration value is to guarantee that the size of an enumeration value is always 4 bytes.

More Help topics

"The FREObject type" on page 22

FREResult

AIR 3.0 and later

The FREResult enumeration defines return values for native extensions C API functions you call.

```
enum FREResult {
    FRE_OK                  = 0,
    FRE_NO_SUCH_NAME        = 1,
    FRE_INVALID_OBJECT      = 2,
    FRE_TYPE_MISMATCH       = 3,
    FRE_ACTIONSCRIPT_ERROR  = 4,
    FRE_INVALID_ARGUMENT    = 5,
    FRE_READ_ONLY           = 6,
    FRE_WRONG_THREAD        = 7,
    FRE_ILLEGAL_STATE       = 8,
    FRE_INSUFFICIENT_MEMORY = 9,
    FREResult_ENUMPADDING   = 0xffff
};
```

The enumeration values have the following meanings:

FRE_OK The function succeeded.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred, and an exception was thrown. The C API functions that can result in this error allow you to specify an FREObject to receive information about the exception.

FRE_ILLEGAL_STATE A call was made to a native extensions C API function when the extension context was in an illegal state for that call. This return value occurs in the following situation. The context has acquired access to an ActionScript BitmapData or ByteArray class object. With one exception, the context can call no other C API functions until it releases the BitmapData or ByteArray object. The one exception is that the context can call `FREInvalidateBitmapDataRect()` after calling `FREAcquireBitmapData()` or `FREAcquireBitmapData2()`.

FRE_INSUFFICIENT_MEMORY The runtime could not allocate enough memory to change the size of an Array or Vector object.

FRE_INVALID_ARGUMENT A pointer parameter is `NULL`.

FRE_INVALID_OBJECT An FREObject parameter is invalid. For examples of invalid FREObject variables, see "FREObject validity" on page 22.

FRE_NO_SUCH_NAME The name of a class, property, or method passed as a parameter does not match an ActionScript class name, property, or method.

FRE_READ_ONLY The function attempted to modify a read-only property of an ActionScript object.

FRE_TYPE_MISMATCH An FREObject parameter does not represent an object of the ActionScript class expected by the called function.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

FREResult_ENUMPADDING This final enumeration value is to guarantee that the size of an enumeration value is always 4 bytes.

Functions you implement

The extensions C API provides signatures of functions that you implement in your extension's native C implementation.

FREContextFinalizer()

AIR 3.0 and later

Signature

```
typedef void (*FREContextFinalizer)(
        FREContext ctx,
);
```

Parameters

ctx The FREContext variable that represents this extension context. See "FREContext" on page 81.

Returns

Nothing.

Description

The runtime calls this function when it disposes of the ExtensionContext instance for this extension context. The following situations cause the runtime to dispose of the instance:

- The ActionScript side calls the `dispose()` method of the ExtensionContext instance.
- The runtime's garbage collector detects no references to the ExtensionContext instance.
- The AIR application is shutting down.

Implement this function to clean up resources specific to this context of the extension. Use the `ctx` parameter to get and then clean up resources associated with native context data and ActionScript context data. See "Context-specfic data" on page 18.

After the runtime calls this function, it calls no other functions of this extension context.

More Help topics

"Extension context initialization" on page 17

"Extension context finalization" on page 18

FREContextInitializer()

AIR 3.0 and later

Signature

```
typedef void (*FREContextInitializer)(
        void*                   extData,
        const uint8_t*           ctxType,
        FREContext               ctx,
        uint32_t*                numFunctionsToSet,
        const FRENamedFunction** functionsToSet
);
```

Parameters

extData A pointer to the extension data of the native extension. The function FREInitializer() created the extension data.

ctxType A string identifying the type of the context. You define this string as required by your extension. The context type can indicate any agreed-to meaning between the ActionScript side and native side of the extension. If your extension has no use for context types, this value can be `Null`. This value is a UTF-8 encoded string, terminated with the null character.

ctx An FREContext variable. The runtime creates this value and passes it to `FREContextInitializer()`. See "FREContext" on page 81.

numFunctionsToSet A pointer to a `unint32_t`. Set `numFunctionsToSet` to a `unint32_t` variable containing the number of functions in the `functionsToSet` parameter.

functionsToSet A pointer to an array of FRNamedFunction elements. Each element contains a pointer to a native function, and the string the ActionScript side uses in the ExtensionContext instance's `call()` method. See "FRENamedFunction" on page 84.

Returns

Nothing.

Description

The runtime calls this method when the ActionScript side calls `ExtensionContext.createExtensionContext()`. Implement this method to do the following:

- Initialize your extension context. The initializations can depend on the context type passed in the `ctxType` parameter.
- Save the value of the `ctx` parameter so it is available to calls the native implementation makes to `FREDispatchStatusEventAsync()`.
- Use the `ctx` parameter to initialize context-specific data. This data includes context-specific native data and context-specific ActionScript data.
- Set up an array of FRENamedFunction objects. Return a pointer to the array in the `functionsToSet` parameter. Return a pointer to the number of array elements in the `numFunctionsToSet` parameter.

The behavior your FREContextInitializer() method can depend on the `ctxType` parameter. The ActionScript side can pass a context type to `ExtensionContext.createExtensionContext()`. Then, the runtime passes the value to `FREContextInitializer()`. This function typically uses the context type to choose the set of methods in the native implementation that the ActionScript side can call. Each context type corresponds to a different set of methods.

More Help topics

"The context type" on page 9

"Extension context initialization" on page 17

FREFinalizer()

AIR 3.0 and later

Signature

```
typedef void (*FREFinalizer)(
        void* extData,
);
```

Parameters

extData A pointer to the extension data of the extension.

Returns

Nothing.

Description

The runtime calls this function when it unloads an extension. However, the runtime does not guarantee that it will unload the extension or call `FREFinalizer()`.

Implement this function to clean up extension resources.

More Help topics

"FREInitializer()" on page 90

"Extension finalization" on page 19

FREFunction()

AIR 3.0 and later

Signature

```
typedef FREObject (*FREFunction)(
        FREContext  ctx,
        void*       functionData,
        uint32_t    argc,
        FREObject   argv[]
);
```

Parameters

ctx The FREContext variable that represents this extension context. See "FREContext" on page 81.

functionData A void*. This pointer points to the data you associated with this FREFunction function in its FRENamedFunction structure. Your implementation of FREContextInitializer() passed a set of FRENamedFunction structures to the runtime in an out parameter. When the runtime calls the FREFunction function, it passes the function this data pointer. See "FRENamedFunction" on page 84.

argc The number of elements in the `argv` parameter.

argv An array of FREObject variables. These pointers correspond to the parameters passed after the function name in the ActionScript call to the ExtensionContext instance's `call()` method.

Returns

An FREObject variable. The default return value is an FREObject for which the type is `FRE_INVALID_OBJECT`.

Description

Implement a function with the FREFunction signature for each native function in the extension. The function name corresponds to the `function` field of an FRENamedFunction element in the array returned in the `functionsToSet` parameter of the `FREContextInitializer()` function.

The runtime calls this FREFunction function when the ActionScript side calls the ExtensionContext instance's `call()` method. The first parameter of `call()` is the same as the `name` field of the FRENamedFunction element. Subsequent parameters to `call()` correspond to the FREObject variables in the `argv` array.

You define the FREFunction to return an FREObject variable. The type of the FREObject variable corresponds to the ActionScript type returned by the `call()` method. If the FREFunction has no return value, the default return value is an FREObject variable with the type `FRE_INVALID_OBJECT`. The default return value results in `call()` returning `null` on the ActionScript side.

Use the `ctx` parameter to:

- Get and set data you associate with the extension. This data can be native context data and ActionScript context data. See "Context-specfic data" on page 18.
- Dispatch an asynchronous event to the ExtensionContext instance on the ActionScript side. See "FREDispatchStatusEventAsync()" on page 95.

Use the functions in "Functions you use" on page 91 to work with the FREObject parameters and the return value, if any.

More Help topics

"FREContextInitializer()" on page 88

"The FREObject type" on page 22

FREInitializer()

AIR 3.0 and later

Signature

```
typedef void (*FREInitializer)(
        void**                  extDataToSet,
        FREContextInitializer*  ctxInitializerToSet,
        FREContextFinalizer*    contextFinalizerToSet
);
```

Parameters

extDataToSet A pointer to a pointer to the extension data of the native extension. Create a data structure to hold extension-specific data. For example, allocate the data from the heap, or provide global data. Set extDataToSet to a pointer to the allocated data.

ctxInitializerToSet A pointer to the pointer to the `FREContextInitializer()` function. Set `ctxInitializerToSet` to the `FREContextInitializer()` function you defined.

ctxFinalizerToSet A pointer to the pointer to the `FREContextFinalizer()` function. Set `ctxFinalizerToSet` to the `FREContextFinalizer()` function you defined. You can set this pointer to `NULL`.

Returns

Nothing.

Description

The runtime calls this method once when it loads an extension. Implement this function to do any initializations that your extension requires. Then set the output parameters.

More Help topics

"FREContextInitializer()" on page 88

"FREContextFinalizer()" on page 87

Functions you use

The native extensions C API provides functions that allow you to access and manipulate ActionScript objects and primitive data.

FREAcquireBitmapData()

AIR 3.0 and later

Usage

```
FREResult FREAcquireBitmapData (
          FREObject      object,
          FREBitmapData*  descriptorToSet
);
```

Parameters

object An FREObject. This FREObject parameter represents an ActionScript BitmapData class object.

descriptorToSet A pointer to a variable of type FREBitmapData. The runtime sets the fields of this structure when the native C implementation calls this method. See "FREBitmapData" on page 82.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The FREBitmapData parameter is set. The ActionScript BitmapData object is available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has already acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases that BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `descriptorToSet` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript BitmapData class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to acquire the bitmap of an ActionScript BitmapData class object. Once you have successfully called this function, you cannot successfully call any other C API function until you call `FREReleaseBitmapData()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the bitmap contents.

After calling this function, you can manipulate the bitmap of the BitmapData object. The bitmap is available in the `descriptorToSet` parameter, along with other information about the bitmap. To notify the runtime that the bitmap or subrectangle of the bitmap has changed, call `FREInvalidateBitmapDataRect()`. When you have finished working with the bitmap, call `FREReleaseBitmapData()`.

More Help topics

"FREReleaseBitmapData()" on page 110

"FREInvalidateBitmapDataRect()" on page 105

FREAcquireBitmapData2()

AIR 3.1 and later

Usage

```
FREResult FREAcquireBitmapData2 (
            FREObject       object,
            FREBitmapData2* descriptorToSet
);
```

Parameters

object An FREObject. This FREObject parameter represents an ActionScript BitmapData class object.

descriptorToSet A pointer to a variable of type FREBitmapData2. The runtime sets the fields of this structure when the native C implementation calls this method. See "FREBitmapData2" on page 83.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The FREBitmapData2 parameter is set. The ActionScript BitmapData object is available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has already acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases that BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `descriptorToSet` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript BitmapData class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to acquire the bitmap of an ActionScript BitmapData class object. Once you have successfully called this function, you cannot successfully call any other C API function until you call `FREReleaseBitmapData()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the bitmap contents.

After calling this function, you can manipulate the bitmap of the BitmapData object. The bitmap is available in the `descriptorToSet` parameter, along with other information about the bitmap. To notify the runtime that the bitmap or subrectangle of the bitmap has changed, call `FREInvalidateBitmapDataRect()`. When you have finished working with the bitmap, call `FREReleaseBitmapData()`.

More Help topics

"FREReleaseBitmapData()" on page 110

"FREInvalidateBitmapDataRect()" on page 105

FREAcquireByteArray()

AIR 3.0 and later

Usage

```
FREResult FREAcquireByteArray (
            FREObject       object,
            FREByteArray*   byteArrayToSet
);
```

Parameters

object An FREObject. This FREObject parameter represents an ActionScript ByteArray class object.

byteArrayToSet A pointer to a variable of type FREByteArray. The runtime sets the fields of this structure when the native C implementation calls this method. See "FREByteArray" on page 84.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The FREByteArray parameter is set. The ActionScript ByteArray object is available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has already acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases that BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `byteArrayToSet` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript ByteArray class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to acquire the bytes of an ActionScript ByteArray class object. Once you have successfully called this function, you cannot successfully call any other C API function until you call `FREReleaseByteArray()`. This prohibition is because other calls could, as a side effect, execute code that invalidates the pointer to the byte array contents.

After calling this function, you can manipulate the bytes of the ByteArray object. The bytes are available in the `byteArrayToSet` parameter, along with the number of bytes. When you have finished working with the bytes, call `FREReleaseByteArray()`.

More Help topics

"FREReleaseByteArray()" on page 111

FRECallObjectMethod()

AIR 3.0 and later

Usage

```
FREResult FRECallObjectMethod(
            FREObject       object,
            const uint8_t*  methodName,
            uint32_t        argc,
            FREObject       argv[],
            FREObject*      result,
            FREObject*      thrownException
);
```

Parameters

object An FREObject that represents the ActionScript class object on which a method is being called.

methodName A uint8_t array. This array is a string that is the name of the method being called. Use UTF-8 encoding for the string and terminate it with the null character.

argc A uint32_t. The value is the number of parameters passed to the method. This parameter is the length of the `argv` array parameter. The value can be 0 when the method to call takes no parameters.

argv[] An FREObject array. Each FREObject element corresponds to the ActionScript class or primitive type passed as a parameter to the method being called. The value can be `NULL` when the method to call takes no parameters.

result A pointer to an FREObject. This FREObject variable is to receive the return value of the method being called. The FREObject variable represents the ActionScript class or primitive type that the method being called returns.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The ActionScript method returned without throwing an exception.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `method` or `result` parameter is `NULL`, or `argc` is greater than 0 but `argv` is `NULL`.

FRE_INVALID_OBJECT The FREObject parameter or an `argv` FREObject element is invalid.

FRE_NO_SUCH_NAME The `methodName` parameter does not match a method of the ActionScript class object that the `object` parameter represents. Another, less likely, reason for this return value exists. Specifically, consider the unusual case when an ActionScript class has two methods with the same name but the names are in different ActionScript namespaces.

FRE_TYPE_MISMATCH The FREObject parameter does not represent an ActionScript class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to call a method of an ActionScript class object.

More Help topics

“The FREObject type” on page 22

FREDispatchStatusEventAsync()

AIR 3.0 and later

Usage

```
FREResult FREDispatchStatusEventAsync(
            FREContext      ctx,
            const uint8_t*  code,
            const uint8_t*  level
);
```

Parameters

ctx An FREContext. This value is the FREContext variable that the extension context received in its context initialization function.

code A pointer to a uint8_t. The runtime sets the `code` property of the StatusEvent object to this value. Use UTF-8 encoding for the string and terminate it with the null character.

level A pointer to a uint8_t. This parameter is a utf8-encoded, null-terminated string. The runtime sets the `level` property of the StatusEvent object to this value.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded.

FRE_INVALID_ARGUMENT The `ctx`, `code`, or `level` parameter is `NULL`. The runtime also returns this value if `ctx` is not valid.

Description

Call this function to dispatch an ActionScript StatusEvent event. The target of the event is the ActionScript ExtensionContext instance that the runtime associated with the context specified by the `ctx` parameter.

Typically, the events this function dispatches are asynchronous. For example, an extension method can start another thread to perform some task. When the task in the other thread completes, that thread calls `FREDispatchStatusEventAsync()` to inform the ActionScript ExtensionContext instance.

***Note:** The `FREDispatchStatusEventAsync()` function is the only C API that you can call from any thread of your native implementation.*

Unless one of its arguments is invalid, `FREDispatchStatusEventAsync()` return `FRE_OK`. However, returning `FRE_OK` does not mean that the event was dispatched. The runtime does not dispatch the event in the following cases:

- The runtime has already disposed of the ExtensionContext instance.
- The runtime is in the process of disposing of the ExtensionContext instance.
- The ExtensionContext instance has no references. It is eligible for the runtime garbage collector to dispose of it.

Set the `code` and `level` parameters to any null-terminated, UTF8-encoded string values. These values are anything you want, but coordinate them with the ActionScript side of the extension.

Example

More Help topics

"FREContext" on page 81

FREGetArrayElementAt()

AIR 3.0 and later

Usage

```
FREResult FREGetArrayElementAt (
            FREObject   arrayOrVector,
            uint32_t    index,
            FREObject*  value
);
```

Parameters

arrayOrVector An FREObject that represents an ActionScript Array or Vector class object.

index A uint32_t that contains the index of the Array or Vector element to get. The first element of an Array or Vector object has index 0.

value A pointer to an FREObject. This method sets the FREObject variable that this parameter points to. The method sets the FREObject variable to correspond to the Array or Vector element at the requested index.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `value` parameter is set to the requested Array or Vector element.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `arrayOrVector` parameter corresponds to an ActionScript Vector object but the `index` is greater than the index of the final element. Another reason for this return value is if the `value` parameter is `NULL`.

FRE_INVALID_OBJECT The `arrayOrVector` FREObject parameter is invalid.

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not represent an ActionScript Array or Vector class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the ActionScript class object or primitive value at the specified index of the ActionScript Array or Vector class object. The FREObject parameter `arrayOrVector` represents the Array or Vector object. The runtime sets the FREObject variable that the `value` parameter points to. It sets the FREObject variable to correspond to the appropriate Array or Vector element.

If an ActionScript Array object does not have a value at the requested index, the runtime sets the FREObject `value` parameter to invalid, but returns `FRE_OK`.

More Help topics

"FRESetArrayElementAt()" on page 111

"FRESetArrayLength()" on page 112

FREGetArrayLength()

AIR 3.0 and later

Usage

```
FREResult FREGetArrayLength (
            FREObject   arrayOrVector,
            uint32_t*   length
);
```

Parameters

arrayOrVector An FREObject that represents an ActionScript Array or Vector class object.

length A pointer to a uint32_t. This method sets the uint32_t variable pointed to by this parameter with the length of the Array or Vector class object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The method set the uint32_t variable that the `length` parameter points to. It sets the variable to the length of the Array or Vector object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `length` parameter is `NULL`.

FRE_INVALID_OBJECT The `arrayOrVector` FREObject parameter is invalid.

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not represent an ActionScript Array or Vector class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the length of an Array or Vector class object. The FREObject parameter `arrayOrVector` represents the Array or Vector object. The runtime returns the length in the uint32_t variable that the `length` parameter points to.

More Help topics

"FRESetArrayLength()" on page 112

"FRESetArrayElementAt()" on page 111

FREGetContextActionScriptData()

AIR 3.0 and later

Usage

```
FREResult FREGetContextActionScriptData( FREContext ctx, FREObject *actionScriptData);
```

Parameters

ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 88.

actionScriptData A pointer to an FREObject variable. When `FREGetContextActionScriptData()` is successful, the runtime sets this parameter. The value corresponds to the ActionScript object previously saved with `FRESetContextActionScriptData()`.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `actionScriptData` parameter corresponds to the ActionScript object previously saved with `FRESetContextActionScriptData()`.

FRE_INVALID_ARGUMENT The `actionScriptData` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get an extension context's ActionScript data.

More Help topics

"FRESetContextActionScriptData()" on page 113

"Context-specfic data" on page 18

FREGetContextNativeData()

AIR 3.0 and later

Usage

```
FREResult FREGetContextNativeData( FREContext ctx, void** nativeData );
```

Parameters

ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 88.

nativeData A pointer to a pointer to the native data.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `nativeData` parameter is set to point to the context's native data.

FRE_INVALID_ARGUMENT The `nativeData` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get an extension context's native data.

More Help topics

"FRESetContextNativeData()" on page 114

"Context-specfic data" on page 18

FREGetObjectAsBool()

AIR 3.0 and later

Usage

```
FREResult FREGetObjectAsBool  ( FREObject object, unint32_t *value );
```

Parameters

object An FREObject.

value A pointer to a uint32_t. The function sets this value to correspond to the value of the Boolean ActionScript variable that the FREObject variable represents. A non-zero value corresponds to `true`. A zero value corresponds to `false`.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a C uint32_t variable to the value of an ActionScript Boolean variable.

More Help topics

"FRENewObjectFromBool()" on page 107

"The FREObject type" on page 22

FREGetObjectAsDouble()

AIR 3.0 and later

Usage

```
FREResult FREGetObjectAsDouble ( FREObject object, double *value );
```

Parameters

object An FREObject.

value A pointer to a double. The function sets this value to correspond to the Boolean, int, or Number ActionScript variable that the FREObject variable represents.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean, int, or Number ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a C double variable to the value of an ActionScript Boolean, int, or Number variable.

More Help topics

"FRENewObjectFromDouble()" on page 107

"The FREObject type" on page 22

FREGetObjectAsInt32()

AIR 3.0 and later

Usage

```
FREResult FREGetObjectAsInt32 ( FREObject object, int32_t *value );
```

Parameters

object An FREObject.

value A pointer to an int32_t. The function sets this value to correspond to the value of the Boolean or int ActionScript variable that the FREObject variable represents.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean or int ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a C int32_t variable to the value of an int or Boolean ActionScript variable.

More Help topics

"FRENewObjectFromInt32()" on page 108

"The FREObject type" on page 22

FREGetObjectAsUint32()

AIR 3.0 and later

Usage

```
FREResult FREGetObjectAsUint32 ( FREObject object, uint32_t *value );
```

Parameters

object An FREObject.

value A pointer to a uint32_t. The function sets this value to correspond to the value of the Boolean or int ActionScript variable that the FREObject variable represents.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` parameter is correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a Boolean or int ActionScript value. An int ActionScript value that is negative also results in this return value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The value parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a C uint32_t variable to the value of an ActionScript Boolean or int variable.

More Help topics

"FRENewObjectFromUint32()" on page 109

"The FREObject type" on page 22

FREGetObjectAsUTF8()

AIR 3.0 and later

Usage

```
FREResult FREGetObjectAsUTF8(FREObject object, uint32_t* length, const uint8_t** value);
```

Parameters

object An FREObject.

length A pointer to a uint32_t. The value of `length` is the number of bytes in the `value` array. The length includes the null terminator. The length corresponds to the length of the String ActionScript variable that the FREObject variable represents.

value A pointer to a uint8_t array. The function fills the array with the characters of the String ActionScript variable that the FREObject variable represents. The string uses UTF-8 encoding terminates with the null character.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `value` and `length` parameters are correctly set.

FRE_TYPE_MISMATCH The FREObject parameter does not contain a String ActionScript value.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_INVALID_ARGUMENT The `value` or `length` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a uint8_t array to the string value of an ActionScript String object.

Consider the following regarding the string returned in the `value` parameter:

- You cannot change the string.

- The string is valid only until the native extension function that the runtime called returns.
- The string becomes invalid if you call any other C API function.

Therefore, to manipulate or access the string later, copy it immediately into your own array.

More Help topics

"FRENewObjectFromUTF8()" on page 109

"The FREObject type" on page 22

FREGetObjectProperty()

AIR 3.0 and later

Usage

```
FREResult FREGetObjectProperty (
                FREObject       object,
                const uint8_t*  propertyName,
                FREObject*      propertyValue,
                FREObject*      thrownException
);
```

Parameters

object An FREObject that represents the ActionScript class object from which to fetch the value of a property.

propertyName A uint8_t array. This array contains a string that is the name of property. Use UTF-8 encoding for the string and terminate it with the null character.

propertyValue A pointer to an FREObject. This method sets this FREObject parameter to represent an ActionScript object that is the requested property.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `propertyValue` parameter is correctly set.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `propertyName` or `propertyValue` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_NO_SUCH_NAME The `propertyName` parameter does not match a property of the ActionScript class object that the `object` parameter represents. Another, less likely, reason for this return value exists. Specifically, consider the unusual case when an ActionScript class has two properties with the same name but the names are in different ActionScript namespaces.

FRE_TYPE_MISMATCH The FREObject parameter does not represent an ActionScript class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the FREObject variable to the data that corresponds to a public property of an ActionScript class object specified by the `object` parameter.

More Help topics

"FRESetObjectProperty()" on page 114

"The FREObject type" on page 22

FREGetObjectType()

AIR 3.0 and later

Usage

```
FREResult FREGetObjectType( FREObject object, FREObjectType *objectType );
```

Parameters

object An FREObject variable.

objectType A pointer to an FREObjectType variable. `FREGetObjectType()` sets this variable to one of the FREObjectType enumeration values.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `objectType` parameter is correctly set.

FRE_INVALID_ARGUMENT The `objectType` parameter is `NULL`.

FRE_INVALID_OBJECT The FREObject parameter is invalid.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to get the FREObjectType enumeration value that best describes an FREObject variable's corresponding ActionScript class object or primitive type.

More Help topics

"FREObjectType" on page 85

"The FREObject type" on page 22

FREInvalidateBitmapDataRect()

AIR 3.0 and later

Usage

```
FREResult FREInvalidateBitmapDataRect(
            FREObject object,
            uint32_t x,
            uint32_t y,
            uint32_t width,
            uint32_t height
);
```

Parameters

object An FREObject that represents a previously acquired ActionScript BitmapData class object.

x A uint32_t. This value is the x coordinate in terms of pixels. The value is relative to the upper-left corner of the bitmap. Along with the `y` parameter, it indicates the upper-left corner of the rectangle to invalidate.

y A uint32_t. This value is the y coordinate in terms of pixels. The value is relative to the upper-left corner of the bitmap. Along with the `x` parameter, it indicates the upper-left corner of the rectangle to invalidate.

width A uint32_t. This value is the width in pixels of the rectangle to invalidate.

height A uint32_t. This value is the height in pixels of the rectangle to invalidate.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The specified rectangle has been invalidated.

FRE_ILLEGAL_STATE The extension context has not acquired the ActionScript BitmapData object.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript BitmapData class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to invalidate a rectangle of an ActionScript BitmapData class object. Before calling this function, call `FREAcquireBitmapData()` or `FREAcquireBitmapData2()`. Call `FREReleaseBitmapData()` after you are done manipulating and invalidating the bitmap.

Invalidating a rectangle of a BitmapData object indicates to the runtime that it will need to redraw the rectangle.

More Help topics

“FREAcquireBitmapData()” on page 91

“FREAcquireBitmapData2()” on page 92

“FREReleaseBitmapData()” on page 110

FRENewObject()

AIR 3.0 and later

Usage

```
FREResult FRENewObject(
            const uint8_t*  className,
            uint32_t        argc,
            FREObject       argv[],
            FREObject*      object,
            FREObject*      thrownException
);
```

Parameters

className A uint8_t array. A string that is the name of the ActionScript class to create an object of. Use UTF-8 encoding for the string and terminate it with the null character.

argc A uint32_t. The number of parameters passed to the constructor of the ActionScript class. This parameter is the length of the `argv` array parameter. The value can be 0 when the constructor takes no parameters.

argv[] An FREObject array. Each FREObject element corresponds to the ActionScript class or primitive type passed as a parameter to the constructor. The value can be `NULL` when the constructor takes no parameters.

object A pointer to an FREObject. When this method returns successfully, this FREObject variable represents the new ActionScript class object.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The `object` parameter represents the new ActionScript class object.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `className` or `object` parameter is `NULL`, or `argc` is greater than 0 but `argv` is `NULL` or empty.

FRE_NO_SUCH_NAME The `className` parameter does not match an ActionScript class name.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an object of an ActionScript class. The constructor that the runtime calls depends on the parameters you pass in the `argv` array.

More Help topics

"The FREObject type" on page 22

FRENewObjectFromBool()

AIR 3.0 and later

Usage

```
FREResult FRENewObjectFromBool ( uint32_t value, FREObject* object);
```

Parameters

value A uint32_t that is the value for a new ActionScript Boolean instance.

object A pointer to an FREObject that points to the data that represents a Boolean ActionScript variable. A non-zero value corresponds to `true`. A zero value corresponds to `false`.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript Boolean instance with the `value` parameter. The runtime sets the FREObject variable to the data corresponding to the new ActionScript instance.

More Help topics

"FREGetObjectAsBool()" on page 99

"The FREObject type" on page 22

FRENewObjectFromDouble()

AIR 3.0 and later

Usage

```
FREResult FRENewObjectFromDouble ( double value, FREObject* object);
```

Parameters

value A double that is the value for a new ActionScript Number instance.

object A pointer to an FREObject that points to the data that represents a Number ActionScript variable.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript Number instance with the `value` parameter. The runtime sets the FREObject variable to the data corresponding to the new ActionScript instance.

More Help topics

"FREGetObjectAsDouble()" on page 100

"The FREObject type" on page 22

FRENewObjectFromInt32()

AIR 3.0 and later

Usage

```
FREResult FRENewObjectFromInt32 ( int32_t value, FREObject* object);
```

Parameters

value An int32_t that is the value for a new ActionScript int instance.

object A pointer to an FREObject that represents an int ActionScript instance.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript int instance with the `value` parameter. The runtime sets the FREObject variable to correspond to the new ActionScript instance.

More Help topics

"FREGetObjectAsInt32()" on page 101

"The FREObject type" on page 22

FRENewObjectFromUint32()

AIR 3.0 and later

Usage

```
FREResult FRENewObjectFromUint32 ( uint32_t value, FREObject* object);
```

Parameters

value A uint32_t that is the value for a new ActionScript int instance with a value greater than or equal to 0.

object A pointer to an FREObject that represents an int ActionScript instance.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The FREObject parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript int instance with the `value` parameter. The runtime sets the FREObject variable to correspond to the new ActionScript int instance.

More Help topics

"FREGetObjectAsUint32()" on page 101

"The FREObject type" on page 22

FRENewObjectFromUTF8()

AIR 3.0 and later

Usage

```
FREResult FRENewObjectFromUTF8(uint32_t length, const uint8_t* value, FREObject* object);
```

Parameters

length A uint32_t that is the length of the string in the `value` parameter, including the null terminator.

value An array of uint8_t elements. The array is the value for the new ActionScript String object. The FREObject variable represents a String ActionScript variable. Use UTF-8 encoding for the string and terminate it with the null character.

object A pointer to an FREObject that points to the data that represents a String ActionScript object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the `object` parameter is correctly set.

FRE_INVALID_ARGUMENT The `object` or `value` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to create an ActionScript String object with the string value specified in `value`. This method sets the FREObject output parameter `object` to correspond to the new ActionScript String instance.

More Help topics

"FREGetObjectAsUTF8()" on page 102

"The FREObject type" on page 22

FREReleaseBitmapData()

AIR 3.0 and later

Usage

```
FREResult FREReleaseBitmapData (FREObject object);
```

Parameters

object An FREObject that corresponds to an ActionScript BitmapData class object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The ActionScript BitmapData object is no longer available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has not acquired the ActionScript BitmapData object.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript BitmapData class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to release an ActionScript BitmapData class object. Before calling this function, call `FREAcquireBitmapData()` or `FREAcquireBitmapData2()` and `FREInvalidateBitmapDataRect()`. After calling this function, you can no longer manipulate the bitmap, but you can again call other native extension C API functions.

More Help topics

"FREAcquireBitmapData()" on page 91

"FREAcquireBitmapData2()" on page 92

"FREInvalidateBitmapDataRect()" on page 105

FREReleaseByteArray()

AIR 3.0 and later

Usage

```
FREResult FREReleaseByteArray (FREObject object);
```

Parameters

object An FREObject that corresponds to an ActionScript ByteArray class object.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The ActionScript ByteArray object is no longer available for you to manipulate.

FRE_ILLEGAL_STATE The extension context has not acquired the ActionScript ByteArray object.

FRE_INVALID_OBJECT The FREObject `object` parameter is invalid.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not correspond to an ActionScript ByteArray object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to release an ActionScript ByteArray class object. Before calling this function, call `FREAcquireByteArray()`. After calling this function, you can no longer manipulate the ByteArray bytes, but you can again call other native extension C API functions.

More Help topics

"FREAcquireByteArray()" on page 93

FRESetArrayElementAt()

AIR 3.0 and later

Usage

```
FREResult FRESetArrayElementAt (
            FREObject   arrayOrVector,
            uint32_t    index,
            FREObject   value
);
```

Parameters

arrayOrVector An FREObject that points to data that represents an ActionScript Array or Vector class object.

index A uint32_t that contains the index of the Array or Vector element to set. The first element of an Array or Vector object has index 0.

value An FREObject. This method sets the Array or Vector element specified by `index` to the ActionScript object represented by the FREObject `value` parameter.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The Array or Vector element is set to the `value` FREOjbect parameter.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_OBJECT The `arrayOrVector` or `value` FREObject parameter is invalid.

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not point to data that represents an ActionScript Array or Vector class object. This return value can also mean that the `arrayOrVector` parameter represents a Vector object and the `value` parameter is not the correct type for that Vector object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the ActionScript class object or primitive value at the specified index of an ActionScript Array or Vector class object. The FREObject parameter `arrayOrVector` corresponds to the Array or Vector object. The FREObject parameter `value` corresponds to the array element value.

More Help topics

"FREGetArrayElementAt()" on page 96

"FREGetArrayLength()" on page 97

FRESetArrayLength()

AIR 3.0 and later

Usage

```
FREResult FRESetArrayLength (
            FREObject   arrayOrVector,
            uint32_t    length
);
```

Parameters

arrayOrVector An FREObject that represents an ActionScript Array or Vector class object.

length A uint32_t. This method sets the length of the Array or Vector class object to this parameter's value.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded. The runtime has changed the size of the Array or Vector object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INSUFFICIENT_MEMORY The runtime could not allocate enough memory to change the size of the Array or Vector object.

FRE_INVALID_ARGUMENT The `length` parameter is greater than 2^{32}.

FRE_INVALID_OBJECT The `arrayOrVector` FREObject parameter is invalid.

FRE_READ_ONLY The `arrayOrVector` FREObject parameter represents a ActionScript Vector object that has a fixed size. (Its `fixed` property is true.)

FRE_TYPE_MISMATCH The `arrayOrVector` FREObject parameter does not represent an ActionScript Array or Vector class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the length of an ActionScript Array or Vector class object. The FREObject parameter `arrayOrVector` corresponds to the Array or Vector object. The runtime changes the size of the Array or Vector object as specified by the `length` parameter.

More Help topics

"FREGetArrayElementAt()" on page 96

"FREGetArrayLength()" on page 97

FRESetContextActionScriptData()

AIR 3.0 and later

Usage

```
FREResult FRESetContextActionScriptData( FREContext ctx, FREObject actionScriptData);
```

Parameters

ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 88.

actionScriptData An FREObject variable.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded.

FRE_INVALID_OBJECT The `actionScriptData` parameter is an invalid FREObject variable.

FRE_INVALID_ARGUMENT An argument is invalid.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set an extension context's ActionScript data.

More Help topics

"FREGetContextActionScriptData()" on page 98

"Context-specfic data" on page 18

FRESetContextNativeData()

AIR 3.0 and later

Usage

```
FREResult FRESetContextNativeData( FREContext ctx, void* nativeData );
```

Parameters

ctx An FREContext variable. The runtime passed this value to `FREContextInitializer()`. See "FREContextInitializer()" on page 88.

nativeData A pointer to the native data.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded.

FRE_INVALID_ARGUMENT The `nativeData` parameter is `NULL`.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set an extension context's native data.

More Help topics

"FREGetContextNativeData()" on page 99

"Context-specfic data" on page 18

FRESetObjectProperty()

AIR 3.0 and later

Usage

```
FREResult FRESetObjectProperty (
                FREObject       object,
                const uint8_t*  propertyName,
                FREObject       propertyValue,
                FREObject*      thrownException
);
```

Parameters

object An FREObject that represents the ActionScript class object for which a property is to be set.

propertyName A uint8_t array. This array contains a string that is the name of property to set. Use UTF-8 encoding for the string and terminate it with the null character.

propertyValue An FREObject that represents the value to set the property to.

thrownException A pointer to an FREObject. If calling this method results in the runtime throwing an ActionScript exception, this FREObject variable represents the ActionScript Error, or Error subclass, object. If no error occurs, the runtime sets this FREObject variable to be invalid. That is, `FREGetObjectType()` for the thrownException FREObject variable returns `FRE_INVALID_OBJECT`. This pointer can be `NULL` if you do not want to handle exception information.

Returns

An FREResult. The possible return values include, but are not limited to, the following:

FRE_OK The function succeeded and the ActionScript class object's property is correctly set.

FRE_ACTIONSCRIPT_ERROR An ActionScript error occurred. The runtime sets the `thrownException` parameter to represent the ActionScript Error class or subclass object.

FRE_ILLEGAL_STATE The extension context has acquired an ActionScript BitmapData or ByteArray object. The context cannot call this method until it releases the BitmapData or ByteArray object.

FRE_INVALID_ARGUMENT The `propertyName` parameter is `NULL`.

FRE_INVALID_OBJECT The `object` or `propertyValue` parameter is an invalid FREObject variable.

FRE_NO_SUCH_NAME The `propertyName` parameter does not match a property of the ActionScript class object that the `object` parameter represents. Another, less likely, reason for this return value exists. Specifically, consider the unusual case when an ActionScript class has two properties with the same name but the names are in different ActionScript namespaces.

FRE_READ_ONLY The property to set is a read-only property.

FRE_TYPE_MISMATCH The FREObject `object` parameter does not represent an ActionScript class object.

FRE_WRONG_THREAD The method was called from a thread other than the one on which the runtime has an outstanding call to a native extension function.

Description

Call this function to set the value of a public property of the ActionScript class object that an FREObject variable represents. Pass the name of the property to set in the `propertyName` parameter. Pass the new property value in the `propertyValue` parameter.

More Help topics

"FREGetObjectProperty()" on page 103

"The FREObject type" on page 22

Chapter 9: Android Java API Reference

For native extension examples using the Android Java API, see Native extensions for Adobe AIR.

Interfaces

The Java API for AIR native extensions defines two interfaces, both of which must be implemented by all extensions.

FREExtension

Package: com.adobe.fre

Runtime version AIR 3

The FREExtension interface defines the interface used by the AIR runtime to instantiate the Java code in a native extension.

Methods

Method	Description
void initialize()	Called by the runtime during extension initialization.
FREContext createContext(String contextType)	Creates an FREContext object.
void dispose()	Called by the runtime when the extension is destroyed.

Every native extension library must implement the FREExtension interface. The fully qualified name of the implementing class must be present in the `<initializer>` element of the extension descriptor file.

For example, if the extension Java code is packaged in a JAR file named *ExampleExtension.jar* and the class implementing FREExtension is *com.example.ExampleExtension*, then you would use the following extension descriptor entry:

```
<platform name="Android-ARM">
    <applicationDeployment>
        <nativeLibrary>ExampleExtension.jar</nativeLibrary>
        <initializer>com.example.ExampleExtension</initializer>
    </applicationDeployment>
</platform>
```

The `<finalizer>` element is not needed when using the Java extension interface.

Method details

createContext

`FREContext createContext( String contextType )`

Creates an FREContext object.

The AIR runtime calls the Java `createContext()` method after the extension invokes the ActionScript `ExtensionContext.createExtensionContext()` method. The runtime uses this context returned for subsequent method invocations.

This function typically uses the `contextType` parameter to choose the set of methods in the native implementation that the ActionScript side can call. Each context type corresponds to a different set of methods. Your extension can create a single context; multiple context instances that provide the same set of features, but with instance-specific state; or multiple context instances that provide different features.

Parameters:

`contextType` A string identifying the type of the context. You define this string as required by your extension. The context type can indicate any agreed-to meaning between the ActionScript side and native side of the extension. If your extension has no use for context types, this value can be `null`. This value is a UTF-8 encoded string, terminated with the null character.

Returns:

`FREContext` the extension context object.

Example:

The following example returns a context object based on the `contextType` parameter. If the `contextType` is the string, “TypeA”, the function returns a unique FREContext instance each time it is called. For other values of `contextType`, the function only creates a single FREContext instance and stores it in the private variable `bContext`.

```
private FREContext bContext;
public FREContext createContext( String contextType ) {
    FREContext theContext = null;
    if( contextType == "TypeA" )
    {
        theContext = new TypeAContext();
    }
    else
    {
        if( bContext == null ) bContext = new TypeBContext();
        theContext = bContext;
    }
    return theContext;
}
```

dispose

`void dispose()`

Use the `dispose()` method to clean up any resources created by this FREExtension implementation. The AIR runtime calls this method when the associated ActionScript ExtensionContext object is disposed or becomes eligible for garbage collection.

initialize

`void initialize()`

Called by the AIR runtime during extension initialization.

Class Example

The following example illustrates a simple FREExtension example that creates a single FREContext object. The example also shows how to use the Android Log class to log informational messages to the Android system log (which you can view using the `adb logcat` command).

```
package com.example;

import android.util.Log;
import com.adobe.fre.FREContext;
import com.adobe.fre.FREExtension;

public class DataExchangeExtension implements FREExtension {

    private static final String EXT_NAME = "DataExchangeExtension";
    private String tag = EXT_NAME + "ExtensionClass";
    private DataExchangeContext context;

    public FREContext createContext(String arg0) {
        Log.i(tag, "Creating context");
        if( context == null) context = new DataExchangeContext();
        return context;
    }

    public void dispose() {
        Log.i(tag, "Disposing extension");
        // nothing to dispose for this example
    }

    public void initialize() {
        Log.i(tag, "Initialize");
        // nothing to initialize for this example
    }
}
```

FREFunction

Package: com.adobe.fre

Runtime version AIR 3

The FREFunction interface defines the interface used by the runtime to invoke the Java functions defined in your native extension.

Methods

Method	Description
FREObject call(FREContext ctx, FREObject[] args)	Called by the AIR runtime when this function is invoked by the ActionScript side of the extension.

Implement the FREFunction interface for each Java function in the extension that can be invoked by the ActionScript portion of the native extension library. Add the class to the Java Map object returned by the `getFunctions()` method of the FREContext instance that provides the function.

The runtime invokes the `call()` method when you execute the ExtensionContext instance’s `call()` method in the ActionScript portion.

When you add the FREFunction to the context’s function map, you specify a string value as a key. Use the same value when invoking the function from ActionScript.

Method details

call

`FREObject call( FREContext ctx, FREObject[] args )`

Called by the runtime when this function is invoked from ActionScript. Implement (or initiate) the functionality provided by an FREFunction instance in the `call()` method.

Parameters:

`ctx` The FREContext variable that represents this extension context.

Use the `ctx` parameter to:

- Get and set data you associate with the extension context.
- Dispatch an asynchronous event to the ExtensionContext instance on the ActionScript side using the FREContext `dispatchStatusEventAsync()` method.

`args` The arguments passed to the function as an array of FREObject variables. The objects in the array are the arguments from the ActionScript ExtensionContext `call()` method used to invoke the Java function.

Returns:

`FREObject` The result. All objects shared between the Java and the ActionScript portions of the extension are encapsulated in an FREObject or one of its subclasses.

Class Example

The following example illustrates a function that takes a string argument and returns a new string with the characters reversed. The `call()` function uses the `ctx` parameter to get an identifier string from the context through which the function is invoked.

```
package com.example;

import com.adobe.fre.FREContext;
import com.adobe.fre.FREFunction;
import com.adobe.fre.FREInvalidObjectException;
import com.adobe.fre.FREObject;
import com.adobe.fre.FRETypeMismatchException;
import com.adobe.fre.FREWrongThreadException;
import android.util.Log;

public class ReverseStringFunction implements FREFunction {
    public static final String NAME = "reverseString";
    private String tag;

    public FREObject call(FREContext arg0, FREObject[] arg1) {
        DataExchangeContext ctx = (DataExchangeContext) arg0;
        tag = ctx.getIdentifier() + "." + NAME;
        Log.i( tag, "Invoked " + NAME );
        FREObject returnValue = null;
        try {
            String value = arg1[0].getAsString();
            value = reverse( value );
            returnValue = FREObject.newObject( value );//Invert the received value

        } catch (IllegalStateException e) {
            Log.d(tag, e.getMessage());
            e.printStackTrace();
        } catch (FRETypeMismatchException e) {
            Log.d(tag, e.getMessage());
            e.printStackTrace();
        } catch (FREInvalidObjectException e) {
            Log.d(tag, e.getMessage());
            e.printStackTrace();
        } catch (FREWrongThreadException e) {
            Log.d(tag, e.getMessage());
            e.printStackTrace();
        }

        return returnValue;
    }

    private String reverse( String source ) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);
        for (i = (len - 1); i >= 0; i--)
            dest.append(source.charAt(i));
            return dest.toString();
    }
}
```

Classes

The Java API for AIR native extensions defines one abstract class, FREContext, that allows you to define the extension context. The API also defines several concrete classes, FREObject and its subclasses, which represent objects shared between the Java and ActionScript sides of a native extension. An extension must implement at least one concrete subclass of FREContext.

FREArray

Package: com.adobe.fre

Inheritance FREObject

Runtime version AIR 3

The FREArray class represents an ActionScript Array or Vector object.

Methods

Method	Description
public static FREArray newArray (String classname, int numElements, boolean fixed)	Creates an ActionScript Vector array object.
public static FREArray newArray (int numElements)	Creates an ActionScript Array object.
public long getLength()	Gets the number of elements in the array.
public void setLength(long length)	Changes the array length.
public FREObject getObjectAt(long index)	Gets the object at the specified index.
public void setObjectAt(long index, FREObject value)	Puts an object into the array at the specified index.

You can work with an FREArray object using the methods defined in the FREArray class, as well as the methods defined in the FREObject class (which is the FREArray's super class). Use the FREObject `getProperty()` and `setProperty()` methods to access or modify the ActionScript-defined properties of the Array and Vector classes. Use `callMethod()` to call the ActionScript-defined methods.

Method details

newArray

`public static FREArray newArray (String classname, int numElements, boolean fixed)`

Creates an ActionScript Vector array object.

Parameters:

`classname` The fully-qualified name of the ActionScript class of the members of the Vector array.

`numElements` The number of elements to allocate for the array.

`fixed` If `true`, the vector length cannot be changed.

Returns:

`FREArray` An FREArray object associated with an ActionScript Vector array object.

Example:

```
FREArray vector = FREArray.newArray( "flash.geom.Matrix3D", 4, true );
```

newArray

`public static FREArray newArray (int numElements)`

Creates an ActionScript Array object.

Parameters:

`numElements` The number of elements to allocate for the array. The elements are undefined.

Returns:

`FREArray` An FREArray object associated with an ActionScript Array object.

Example:

```
FREArray array = FREArray.newArray( 4 );
```

getLength

`public long getLength()`

Gets the number of elements in the array.

Returns:

`long` The length of the array.

Example:

```
long length = asArray.getLength();
```

setLength

`public void setLength( long length )`

Changes the length of this array. If the new length is shorter than the current length, the array is truncated.

Parameters:

`length` The new length for the array.

Example:

```
asArray.setLength( 4 );
```

getObjectAt

`public FREObject getObjectAt( long index )`

Gets an element from the array.

Parameters:

index The position of the element to retrieve. (0-based)

Returns:

`FREObject` The FREObject instance associated with the ActionScript object in the array.

Example:

```
FREObject element = asArray.getObjectAt( 2 );
```

setObjectAt

`public void setObjectAt( long index, FREObject value )`

Puts an object into the array at the specified index.

Parameters:

`index` The position in the array at which to put the object. (0-based)

`value` An FREObject containing the primitive value or ActionScript object to insert.

Example:

```
FREObject stringElement = FREObject.newObject("String element value");
FREArray asVector = FREArray.newArray( "String", 1, false );
asVector.setObjectAt( 0, stringElement );
```

FREByteArray

Package: com.adobe.fre

Inheritance FREObject

Runtime version AIR 3

The FREByteArray class represents an ActionScript ByteArray object.

Methods

Method	Description
public static FREByteArray newByteArray()	Creates an empty ActionScript ByteArray object.
public long getLength()	Returns the length of the byte array in bytes.
public ByteBuffer getBytes()	Gets the contents of the ActionScript ByteArray object as a Java ByteBuffer.
public void acquire()	Acquires a lock on the ActionScript object.
public void release()	Releases a lock on the ActionScript object.

Access the data in the ByteArray object by calling `getBytes()`. Before accessing the byte data in an array referenced by ActionScript, you must call `acquire()` to lock the object. After you are done accessing or modifying the data, call `release()` to release the lock.

While you have a lock on the array, you can modify the existing data in the buffer, but you cannot change the size of the array. To modify the array size, release the lock and change the ActionScript-defined `length` property using the `setProperty()` method (defined by the FREObject super class). You can use the `getProperty()`, `setProperty()`, and `callMethod()` functions to access any of the properties and methods defined by the ActionScript ByteArray class.

Method details

newByteArray

`public static FREByteArray newByteArray()`

Creates an empty ActionScript ByteArray object and its associated Java FREByteArray instance.

Returns:

`FREByteArray` The FREByteArray object representing an ActionScript ByteArray.

Example:

```
FREBytearray bytearray = FREByteArray.newByteArray();
```

acquire

`public void acquire()`

Acquires a lock on this object so that the data cannot be modified or disposed by the runtime or application code while you are accessing it. When you have a lock, you cannot read or modify any of the ActionScript-defined properties of the object.

getLength

`public long getLength()`

Returns the number of bytes in the byte array. You must call this object's `acquire()` function before calling this method.

Returns:

`long` The number of bytes in the byte array.

getBytes

`public ByteBuffer getBytes()`

Returns the byte data in the array. You must call `acquire()` to lock the object before calling this method. The buffer is only valid while you have a lock.

`java.nio.ByteBuffer` The data in the byte array.

Example:

```
FREByteArray bytearray = FREByteArray.newByteArray();
bytearray.acquire();
ByteBuffer bytes = bytearray.getBytes();
bytes.putFloat( 16.3 );
bytearray.release();
```

release

`public void release()`

Releases the lock obtained by `acquire()`. You must release the lock before accessing any FREByteArray properties other than the data returned by `getBytes()`.

FREBitmapData

Package: com.adobe.fre

Inheritance FREObject

Runtime version AIR 3

The FREBitmapData class represents an ActionScript BitmapData object.

Methods

Method	**Description**
public static FREBitmapData newBitmapData (int width, int height, boolean transparent, Byte[] fillColor)	Creates an ActionScript BitmapData object.
public int getWidth()	Gets the bitmap width.
public int getHeight()	Gets the bitmap height.
public boolean hasAlpha()	Indicates whether the bitmap has an alpha channel.
public boolean isPremultiplied()	Indicates whether the bitmap colors are already multiplied by the alpha value.
public int getLineStride32()	Gets the number of 32-bit values per bitmap scan line.
ByteBuffer getBits()	Gets the bitmap pixels.
public void acquire()	Acquires a lock on the ActionScript object.
void invalidateRect(int x, int y, int width, int height)	Marks a rectangle within the bitmap as in need of update.
public void release()	Releases a lock on the ActionScript object.
Added in AIR 3.1:	
public boolean isInvertedY	Indicates the order in which the rows of image data are stored.

Most properties of an FREBitmapData object are read only. For example, you cannot change the width or height of a bitmap. (To change these properties, create a new FREBitmapData object with the desired dimensions.) You can, however, access and modify an existing bitmap's pixel values using the `getBits()` method. You can also access the ActionScript-defined properties of the BitmapData object using the FREObject `getProperty()` and `setProperty()` methods and call the object's ActionScript methods with the FREObject `callMethod()` function.

Before calling `getBits()` on FREBitmapData object that is referenced by ActionScript code, lock the bitmap using the `acquire()` method. This prevents the AIR runtime or the application from modifying or disposing the bitmap object while your function is running (possibly in another thread). After modifying the bitmap, call `invalidateRect()` to notify the runtime that the bitmap has changed and release the lock with `release()`. You cannot use ActionScript-defined properties and methods while you have acquired a lock.

The AIR runtime defines pixel data as a 32-bit value containing three color components, plus alpha in the order: ARGB. Each component within the data is one byte. Valid bitmap data is stored with the color components premultiplied by the alpha component. (However, it is possible to receive a technically invalid BitmapData in which the colors have not been premultiplied. For example, rendering bitmaps with the Context3D class can produce such an invalid bitmap. In this case, the `isPremultiplied()` method still reports `true`.)

Method details

newBitmapData

```
public static FREBitmapData newBitmapData (int width, int height, boolean transparent, Byte[]
fillColor)
```

Creates an FREBitmapData object, which can be returned to the ActionScript side of the extension as a BitmapData instance.

Parameters:

`width` The width in pixels. In AIR 3+, there is no arbitrary limit to the width or height of a bitmap (other than the maximum value of an ActionScript signed integer). However, practical memory limits still apply. Bitmap data occupies 32 bits of memory per pixel.

`height` The height in pixels.

`transparent` Specifies whether this bitmap uses the alpha channel. This parameter corresponds to the `hasTransparency()` method of the created FREBitmapData and the `transparent` property of the ActionScript BitmapData object.

`fillColor` The 32-bit, ARGB color value with which to fill the new bitmap. If the `transparent` parameter is `false`, the alpha component of the color is ignored.

Returns:

`FREBitmapData` The FREBitmapData object associated with an ActionScript BitmapData object.

Example:

```
Byte[] fillColor = {0xf,0xf,0xf,0xf,0xf,0xf,0xf,0xf};
FREBitmapData bitmap = FREBitmapData.newBitmapData( 320, 200, true, fillColor );
```

getWidth

```
public int getWidth()
```

Returns the width, in pixels, of the bitmap. This value corresponds to the `width` property of the ActionScript BitmapData class object. You must call this object's `acquire()` function before calling this method.

Returns:

`int` the number of pixels across the horizontal axis of the bitmap.

getHeight

```
public int getHeight()
```

The height, in pixels, of the bitmap. This value corresponds to the `height` property of the ActionScript BitmapData class object. You must call this object's `acquire()` function before calling this method.

Returns:

`int` The number of pixels across the vertical axis of the bitmap.

hasAlpha

```
public boolean hasAlpha()
```

Indicates whether the bitmap supports per-pixel transparency. This value corresponds to the `transparent` property of the ActionScript BitmapData class object. If the value is `true`, then the alpha component of the pixel color values is used. If the value is `false`, the alpha component of the color is not used. You must call this object's `acquire()` function before calling this method.

Returns:

`boolean` true if the bitmap uses an alpha channel.

isPremultiplied

```
public boolean isPremultiplied()
```

Indicates whether the bitmap pixels are stored as premultiplied color values. A true value means the red, green, and blue components of a pixel color *are* premultiplied by the alpha component. ActionScript BitmapData objects are currently always premultiplied (but future changes to the runtime could make it possible to create a bitmap that is not premultiplied). For more information about premultiplied color values, see BitmapData.getPixel() in the *ActionScript 3.0 Reference for the Adobe Flash Platform*. You must call this object's `acquire()` function before calling this method.

Returns:

`boolean` `true` if the pixel color components are already multiplied by the alpha component.

getLineStride32

`public int getLineStride32()`

Specifies the amount of data in each horizontal line of the bitmap. Use this value with the byte array returned by `getBits()` to parse the image. For example, the position of the pixel immediately below a pixel located at position `n` in the byte array is `n + getLineStride32()`. You must call this object's `acquire()` function before calling this method.

Returns:

`int` The amount of data for each horizontal line in the image.

Example:

The following example moves the position of the byte array containing the pixels colors to the beginning of the third line in a bitmap:

```
Byte[] fillColor = {0xf,0xf,0xf,0xf,0xf,0xf,0xf,0xf};
FREBitmapData bitmap = FREBitmapData.newBitmapData( 320, 200, true, fillColor );
int lineStride = bitmap.getLineStride32();
bitmap.acquire();
ByteBuffer pixels = bitmap.getBits();
pixels.position( lineStride * 3 );
//do something with the image data
bitmap.release();
```

getBits

`ByteBuffer getBits()`

Returns an array of pixel color values. You must call `acquire()` before using this method and call `release()` when you are done modifying the data. If you make changes to the pixel data, call `invalidateRect()` to notify the AIR runtime that the bitmap has changed. Otherwise, the displayed graphics are not updated.

Returns:

`java.nio.ByteBuffer` The image data.

Example:

```
FREBitmapData bitmap = FREBitmapData.newBitmapData( 320, 200, true, fillColor );
bitmap.acquire();
ByteBuffer pixels = bitmap.getBits();
//do something with the image data
bitmap.release();
```

acquire

`public void acquire()`

Acquires a lock on the image so that the data cannot be modified or disposed by the runtime or application code while you are accessing it. You only need to acquire a lock for bitmaps passed to the function from the ActionScript side of the extension.

While you have a lock, you can only access the data returned by the `getBits()` function. Accessing or attempting to modify the other properties of the FREBitmapData results in an exception.

invalidateRect

```
void invalidateRect( int x, int y, int width, int height )
```

Notifies the runtime that you have changed a region of the image. The screen display of a bitmap data object is not updated unless you call this method. You must call this object's `acquire()` function before calling this method.

Parameters:

`x` The upper-left corner of the rectangle to invalidate.

`y` the upper-left corner of the rectangle to invalidate.

`width` the width of the rectangle to invalidate.

`height` the height of the rectangle to invalidate.

release

```
public void release()
```

Releases the lock obtained by `acquire()`. You must release the lock before accessing any bitmap properties other than the pixel data returned by `getBits()`.

isInvertedY

```
public boolean isInvertedY()
```

Indicates the order in which the rows of image data are stored. If `true`, then the last row of image data appears first in the buffer returned by the `getBits()` method. Images on Android are normally stored starting with the first row of data, so this property is typically `false` on the Android platform.

Added in AIR 3.1.

FREContext

Package: com.adobe.fre

Inheritance java.lang.Object

Runtime version AIR 3

The FREContext class represents a Java execution context defined by an AIR native extension.

Methods

Method	**Description**
public abstract Map<String,FREFunction> getFunctions()	Called by the AIR runtime to discover the functions provided by this context.
public FREObject getActionScriptData()	Gets any data in the `actionScriptData` property of the ActionScript ExtensionContext object associated with this FREContext.
public void setActionScriptData(FREObject)	Sets the `actionScriptData` property of the ActionScript ExtensionContext object associated with this FREContext.
public abstract void dispose()	Called by the AIR runtime when the ActionScript ExtensionContext object associated with this FREContext instance is disposed.
public android.app.Activity getActivity()	Returns the application Activity.
public native int getResourceId(String resourceString)	Returns the ID for an Android resource.
public void dispatchStatusEventAsync(String code, String level)	Dispatches a StatusEvent to the application via the ExtensionContext object associated with this FREContext instance.

Your extension must define one or more concrete subclasses of the FREContext class. The `createContext()` method in your implementation of the FREExtension class creates instances of these classes when the ActionScript side of the extension invokes the `ExtensionContext.createExtensionContext()` method.

You can organize the contexts provided by an extension in various ways. For example, you could create a single instance of a context class that persists for the lifetime of the application. Alternately, you could create a context class that is closely bound to a set of native objects with limited lifespan. You could then create a separate context instance for each set of these objects and dispose the context instance along with the associated native objects.

***Note:** If you have called the acquire() method of any FREByteArray or FREBitmapData object, then you cannot call the methods defined by the FREObject class of any object.*

Method details

getFunctions

`public abstract Map<String,FREFunction> getFunctions()`

Implement this function to return a Java Map instance that contains all the functions exposed by the context. The map key value is used to look up the FREFunction object when a function is invoked.

Returns:

`Map<String, FREFunction>` A Map object containing a string key value and a corresponding FREFunction object. The AIR runtime uses this Map object to look up the function to invoke when the ActionScript part of the extension invokes the `call()` method of the ExtensionContext class.

Example:

The following getFunctions() example creates a Java HashMap containing three hypothetical functions. You can invoke these functions in the ActionScript code of the extension by passing the string, “negate”, “invert”, or “reverse” to the ExtensionContext `call()` method.

```
@Override
    public Map<String, FREFunction> getFunctions() {
        Map<String, FREFunction> functionMap = new HashMap<String, FREFunction>();
        functionMap.put( "negate", new NegateFunction() );
        functionMap.put( "invert", new InvertFunction() );
        functionMap.put( "reverse", new ReverseFunction() );
        return functionMap;
}
```

getActionScriptData

`public FREObject getActionScriptData()`

Gets the ActionScript data object associated with this context. This data object can be read and modified by both Java and ActionScript code.

Returns:

`FREObject` An FREObject representing the data object.

Example:

```
FREObject data = context.getActionScriptData();
```

setActionScriptData

public void setActionScriptData(FREObject object)

Sets the ActionScript data object associated with this context. This data object can be read and modified by both Java and ActionScript code.

Parameters:

`value` An FREObject containing a primitive value or an ActionScript object.

Example:

```
FREObject data = FREObject.newObject( "A string" );
context.setActionScriptData( data );
```

getActivity

`public android.app.Activity getActivity()`

Gets a reference to the application Activity object. This reference is required by many APIs in the Android SDK.

Returns:

`android.app.Activity` The Activity object of the AIR application.

Example:

```
Activity activity = context.getActivity();
```

getResourceID

`public native int getResourceId( String resourceString )`

Looks up the resource ID of a native Android resource.

Specify the string identifying the resource in the `resourceString` parameter. Follows the naming conventions for resource access on Android. For example a resource that you would access in Android native code as:

```
R.string.my_string
```

would have the following `resourceString`:

```
"string.my_string"
```

This method will throw Resources.NotFoundException if the resource specified by `resourceString` cannot be found.

Parameters:

`resourceString` A string identifying the Android resource.

Returns:

`int` The ID of the Android resource.

Example:

```
int myResourceID = context.getResourceID( "string.my_string" );
```

dispatchStatusEventAsync

```
public void dispatchStatusEventAsync( String code, String level )
```

Call this function to dispatch an ActionScript StatusEvent event. The target of the event is the ActionScript ExtensionContext instance that the runtime associated with this FREContext object.

Typically, the events this function dispatches are asynchronous. For example, an extension method can start another thread to perform some task. When the task in the other thread completes, that thread calls `dispatchStatusEventAsync()` to inform the ActionScript ExtensionContext instance.

***Note:** The `dispatchStatusEventAsync()` function is the only Java API that you can call from any thread of your native implementation.*

The runtime does not dispatch the event in the following cases:

- The runtime has already disposed of the ExtensionContext instance.
- The runtime is in the process of disposing of the ExtensionContext instance.
- The ExtensionContext instance has no references. It is eligible for the runtime garbage collector to dispose of it.

Set the `code` and `level` parameters to any string values. These values can be anything you want, but coordinate them with the ActionScript side of the extension.

Parameters:

`code` A description of the status being reported.

`level` The extension-defined category of message.

Example:

```
context.dispatchStatusEventAsync( "Processing finished", "progress" );
```

dispose

```
public abstract void dispose()
```

Called by the AIR runtime when an associated ActionScript ExtensionContext object is disposed and this FREContext object has no other references. You can implement this method to clean up context resources.

Class Example

The following example creates a subclass of FREContext to return context information:

```
package com.example;
import java.util.HashMap;
import java.util.Map;
import android.util.Log;
import com.adobe.fre.FREContext;
import com.adobe.fre.FREFunction;

public class DataExchangeContext extends FREContext {
    private static final String CTX_NAME = "DataExchangeContext";
    private String tag;

    public DataExchangeContext( String extensionName ) {
        tag = extensionName + "." + CTX_NAME;
        Log.i(tag, "Creating context");
    }
    @Override
    public void dispose() {
        Log.i(tag, "Dispose context");
    }
    @Override
    public Map<String, FREFunction> getFunctions() {
     Log.i(tag, "Creating function Map");
     Map<String, FREFunction> functionMap = new HashMap<String, FREFunction>();
     functionMap.put( NegateBooleanFunction.NAME, new NegateBooleanFunction() );
     functionMap.put( InvertBitmapDataFunction.NAME, new InvertBitmapDataFunction() );
     functionMap.put( InvertByteArrayFunction.NAME, new InvertByteArrayFunction() );
     functionMap.put( InvertNumberFunction.NAME, new InvertNumberFunction() );
     functionMap.put( ReverseArrayFunction.NAME, new ReverseArrayFunction() );
     functionMap.put( ReverseStringFunction.NAME, new ReverseStringFunction() );
     functionMap.put( ReverseVectorFunction.NAME, new ReverseVectorFunction() );

    return functionMap;
    }

    public String getIdentifier()
    {
        return tag;
    }
}
```

FREObject

Package: com.adobe.fre

Inheritance java.lang.Object

Subclasses FREArray, FREBitmapData, FREByteArray

Runtime version AIR 3

The FREObject class represents an ActionScript object to Java code.

Methods

Method	Description
public static FREObject newObject(int value)	Creates an FREObject containing a 32-bit signed integer value.
public static FREObject newObject(double value)	Creates an FREObject that contains a Java double value, which corresponds to the ActionScript Number type.
public static FREObject newObject(boolean value)	Creates an FREObject that contains a boolean value.
public static FREObject newObject(String value)	Creates an FREObject that contains a string value.
public int getAsInt()	Accesses the data in the FREObject as a Java int value.
public double getAsDouble()	Accesses the data in the FREObject as a Java double value.
public Boolean getAsBool()	Accesses the data in the FREObject as a Java boolean value.
public String getAsString()	Accesses the data in the FREObject as a Java String value.
public static native FREObject newObject(String className, FREObject[] constructorArgs)	Creates an FREObject referencing a new instance of an ActionScript class.
public FREObject getProperty(String propertyName)	Gets the value of an ActionScript property.
public void setProperty(String propertyName, FREObject propertyValue)	Sets the value of an ActionScript property.
public FREObject callMethod(String methodName, FREObject[] methodArgs)	Invokes an ActionScript method.

Use FREObjects to share data between the Java and the ActionScript code in an extension. The runtime associates an FREObject variable with the corresponding ActionScript object. You can access the properties and methods of the ActionScript object associated with an FREObject using the `getProperty()`, `setProperty()`, and `callMethod()` functions.

An FREObject is a general purpose object. It can represent a primitive value or a class type. If you access the object in a way that is incompatible with the data in the object, an FRETypeMismatchException is thrown.

FREObjects can be used to represent any ActionScript object. In addition, the subclasses FREArray, FREBitmapData, and FREByteArray provide additional methods for handling specific kinds of data.

You can pass data from Java to ActionScript code by returning an FREObject from the `call()` method of an FREFunction instance. The ActionScript code receives the data as an ActionScript Object, which can be cast to the appropriate primitive or class type. You can also set the properties and call the methods of an ActionScript object for which you have an FREObject reference. When setting a property or the parameter of an ActionScript method, you use an FREObject.

You can pass data from ActionScript to Java code as the parameter to the `call()` method of an FREFunction instance. The argument is encapsulated in an FREObject instance when the `call()` method is invoked. If the argument is a reference to an ActionScript object, the Java code can read the property values and call the methods of that object. These properties and function return values are provided to your Java code as FREObjects. You can use the FREObject methods to access the data as Java types and, when appropriate, downcast the object to an FREObject subclass such as FREBitmapData.

Method details

newObject(int)

```
public static FREObject newObject( int value )
```

Creates an FREObject containing a 32-bit signed integer value.

Parameters:

`value` A signed integer.

Returns:

`FREObject` An FREObject.

Example:

```
FREObject value = FREObject.newObject( 4 );
```

newObject(double)

```
public static FREObject newObject( double value )
```

Creates an FREObject that contains a Java double value, which corresponds to the ActionScript Number type.

Parameters:

`value` A double.

Returns:

`FREObject` An FREObject.

Example:

```
FREObject value = FREObject.newObject( 3.14156d );
```

newObject(boolean)

```
public static FREObject newObject( boolean value )
```

Creates an FREObject that contains a boolean value.

Parameters:

`value` true or false.

Returns:

`FREObject` An FREObject.

Example:

```
FREObject value = FREObject.newObject( true );
```

newObject(String)

```
public static FREObject newObject( String value )
```

Creates an FREObject that contains a string value.

Parameters:

`value` A string.

Returns:

`FREObject` An FREObject.

Example:

```
FREObject value = FREObject.newObject( "A string value" );
```

getAsInt
`public int getAsInt()`

Accesses the data in the FREObject as a Java int value.

Returns:

`int` The integer value.

Example:

```
int value = FREObject.getAsInt();
```

getAsDouble
`public double getAsDouble()`

Accesses the data in the FREObject as a Java double value.

Returns:

`double` The double value.

Example:

```
double value = FREObject.getAsInt();
```

getAsBool
`public Boolean getAsBool()`

Accesses the data in the FREObject as a Java boolean value.

Returns:

`boolean` `true` or `false`

Example:

```
boolean value = FREObject.getAsInt();
```

getAsString
`public String getAsString()`

Accesses the data in the FREObject as a Java String value.

Returns:

`String` The string value.

Example:

```
String value = FREObject.getAsInt();
```

newObject(String, FREObject[])
`public static native FREObject newObject( String className, FREObject[] constructorArgs )`

Creates an FREObject referencing a new instance of an ActionScript class.

Parameters:

`className` The fully-qualified ActionScript class name.

`constructorArgs` The arguments to pass to the ActionScript class constructor as an array of FREObjects. Set to `null` if the class constructor has no parameters.

Returns:

`FREObject` An FREObject representing a new instance of an ActionScript class.

Example:

```
FREObject matrix = FREObject.newObject( "flash.geom.Matrix", null );
```

getProperty

```
public FREObject getProperty( String propertyName )
```

Gets the value of an ActionScript property.

Parameters:

`propertyName` The name of the property to access.

Returns:

`FREObject` The value of the specified property as an FREObject.

Example:

```
FREObject isDir = fileobject.getProperty( "isDirectory" );
```

setProperty

```
public void setProperty( String propertyName, FREObject propertyValue )
```

Sets the value of an ActionScript property.

Parameters:

`propertyName` The name of the property to set.

`propertyValue` An FREObject containing the new property value.

Example:

```
fileobject.setProperty( "url", FREObject.newObject( "app://file.txt" ) );
```

callMethod

```
public FREObject callMethod( String methodName, FREObject[] methodArgs )
```

Invokes an ActionScript method.

Parameters:

`methodName` The name of the method to invoke.

`methodArgs` An array of FREObjects containing the arguments for the method in the order in which the method parameters are declared.

Returns:

`FREObject` The method result. If the ActionScript method returns an Array, Vector, or BitmapData object, you can cast the result to an appropriate subclass of FREObject.

Example:

```
FREObject[] args = new FREObject[1]
args[0] = FREObject.newObject( "assets/image.jpg" );
FREObject imageFile = directoryobject.callMethod( "resolvePath", args );
```